High Performance Computing | RWTH AACHEN UNIVERSITY

Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

Verteiltes Online-Training von datenbasierten Verbrennungsmodellen in Simulationen reaktiver Strömungen

Distributed Online Training of Data-Driven Combustion Models in Reactive Flow Simulations

Masterarbeit

Tim Andres
Matrikelnummer: 380096

Aachen, den 2. April 2026

Erstgutachter: Prof. Dr. rer. nat. Matthias S. Müller (')

Zweitgutachter: Prof. Dr.-Ing Heinz G. Pitsch (*)

Betreuer: Fabian Orland, M.Sc. (')
Ludovico Nista, M.Sc. (*)

(') Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Lehrstuhl und Institut für Technische Verbrennung, RWTH Aachen University

Communicated by Prof. Dr. rer. nat. Matthias S. Müller

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 2. April 2026

Kurzfassung

Diese Arbeit behandelt das Trainieren von Machine Learning Modellen basierend auf Daten von hocheffizienter Simulationssoftware. Da das Trainieren von solchen Modell viele Epochen und Datenproben benötigt um qualitative Ergebnisse zu erzielen, ist es wichtig zu garantieren, dass der Prozess nicht durch Einschränkungen in der Bandbreite gehindert ist. Der gängige Ablauf ist, dass die Daten in einem ersten Schritt von der Simulation erstellt und auf einen Datenträger gespeichert werden. Im zweiten Schritt werden diese Daten dann erneut gelesen und von dem Modell verarbeitet. Häufig laufen beide Schritt in direkter Reihenfolge ab, sodass es eine Belastung des Datenträgers nicht notwendig ist. Zusätzlich kann man Datenvolumen erzeugen, die mehrere Terabyte umfassen und somit ohnehin nicht mehr auf jedes beliebige Medium geschrieben werden können. Hier werden dann Kapazität und Datenrate für Lese- und Schreibvorgänge zu einem Engpass, der das Trainieren unnötigerweise behindet. Es ist das Ziel dieser Arbeit, aufbauen von existieren Ansätzen, eine Lösung zu diesem Problem zu finden und es konkret auf Wasserstoffverbrennungs-Simulationen anzuwenden. Die bisherigen Softwarekomponenten rund um der Simulationssoftware CIAO und der Bibliothek AIxeleratorService sollen erweitert werden um die Fähigkeiten zu erlangen Modelle auf den Daten zu trainieren. Das Konzept wird in die bestehende Architektur integriert und seine Funktionstüchtigkeit vorgestellt. In der anschließenden Auswertung zeigt sich, dass im Vergleich zu traditionellen Ansätzen neue Effekte auftreten können, die das Lernen beeinträchtigen können. Zusätzlich findet eine genauere Analyse der Eingabedaten heraus, dass ein Großteil der Simulationsdaten wenig bis gar nichts zu der Qualität des Modells beiträgt und nicht berücksichtigt werden muss. Diese Datenpunkte lassen sich vor dem Training über ihre Varianz herausfiltern, sodass hierfür keine GPU Zeit verwendet werden muss. In einer abschließenden Beurteilung der Laufzeit stellt sich des Weiteren heraus, dass, obwohl unsere Umsetzung eine gute Laufzeit erzielt, die Gesamtlaufzeit stark durch die Simulationssoftware dominiert wird.

Stichwörter: HPC, MPI, Machine Learning, Wasserstoff, Simulation

Abstract

This thesis deals with training machine learning models based on data from highly efficient simulation software. Since training such models requires many epochs and data samples to achieve qualitative results, it is important to ensure that the process is not hindered by bandwidth limitations. The usual procedure is that, in a first step, the data is created by the simulation and stored on a storage device. In a second step, this data is then read again and processed by the model. Often, both steps run in a direct sequence, such that there is no need to read from the storage device. In addition, data volumes can be generated that comprise several terabytes and thus can no longer be written to any medium. In this case, the capacity and data rate for read and write operations become a bottleneck that unnecessarily hinders training. The goal of this work is to build on existing approaches to find a solution to this problem and apply it specifically to hydrogen combustion simulations. The existing software components related to the CIAO simulation software and the AIxeleratorService library are expanded to include the ability to train models on the data. The concept will be integrated into the existing architecture and its functionality will be presented. The following evaluation shows that, compared to traditional approaches, new effects can occur that can impair learning. In addition, a more detailed analysis of the input data reveals that a large part of the simulation data contributes little or nothing to the quality of the model and does not need to be taken into account. These data points can be filtered out before training based on their variance, so that no GPU time needs to be used for this. In a final assessment of the runtime, it also turns out that, although our implementation achieves a good runtime, the overall runtime is strongly dominated by the simulation software.

Keywords: HPC, MPI, Machine Learning, Hydrogen, Simulation

Contents

List of Figures xi

List of Tables xv

1. Introduction 3
- 1.1. The Limits of Model Training . . . 3
- 1.2. Outline & Contributions . . . 4

2. Real World Data 7
- 2.1. Chemical Simulations . . . 7
  - 2.1.1. Simulating using Neural Networks . . . 8
- 2.2. MNIST . . . 10
- 2.3. A Model for MNIST . . . 12
- 2.4. Training the Model on MNIST . . . 13

3. Training a Model 17
- 3.1. How to Train Your Model . . . 17
  - 3.1.1. Gradient Descent Methods . . . 18
  - 3.1.2. Backpropagation . . . 19
  - 3.1.3. Basic Training Loop . . . 21

4. Enabling Technologies 25
- 4.1. An Overview . . . 25
- 4.2. CIAO . . . 26
- 4.3. ML-Interface . . . 26
- 4.4. AIxeleratorService . . . 27

5. Distributed & Online 33
- 5.1. Training in a Distributed Setting . . . 33
- 5.2. Training with Online Data . . . 34
- 5.3. Existing Implementations . . . 35
  - 5.3.1. Melissa-DL . . . 35
  - 5.3.2. River . . . 37
  - 5.3.3. Horovod . . . 37

6. Training using AIxeleratorService 39
- 6.1. Back to MNIST . . . 39
- 6.2. Combustion Models . . . 40

6.3. The TrainingStrategy 42
6.4. Ensuring Integrity 44

7. Analysis & Evaluation **47**
7.1. Model Performance 47
7.2. Computational Performance 51

8. Conclusion **57**
8.1. Outlook 58

A. Appendix **61**
A.1. Hardware used for Benchmarks 61

Bibliography **65**

LLM Usage **73**

List of Figures

2.1. Examples for data that is produced by a simulation.[21] The colors indicate the H_2O mass fraction with bright colors representing more H_2O and darker colors less. All samples have the dimensions `64x64`. . . 8
2.2. The original UNet architecture 10
2.3. The UNet++ model 11
2.4. The results of our first experiments with the UNet++ on simulated combustion data from CIAO. The model was trained on 50 epochs in figures (a) and (b) and on 200 epochs in figures (c) and (d). The training process utilizes 20000 samples. 12
2.5. A visual representation of the MNIST dataset. [40]The pictures were directly created from the dataset itself. Each pixel is encoded as a floating point value that represents its brightness. 13
2.6. A visual representation of the model we train on MNIST. 14
2.7. The tenth run training the model on MNIST. It is representative for all iterations we did when training MNIST. The loss is computed on the test dataset after each epoch. It is clearly visible that we get a significant improvement throughout the entire training process. The loss we get on the training set during our entire run. We can see that the loss is starting much higher than on the test set. 16
2.8. The loss across all 30 runs with a mean value of around 0.0292345 Note the scaling on the y-axis. The runtime across all 30 runs with a mean value of 94.54 seconds. 16

3.1. Some commonly used losses available in PyTorch[57] 18
3.2. Backprop visualization 20
3.4. PyTorch inspired sample code for training a model. 21
3.3. PyTorch build an internal graph for each tensor that records all layers that were applied to it.[57, 6] 21
3.5. The training cycle 22

4.1. An abstract view of the inference process. Function parameters are omitted for readability. In the inference case, each call usually passes the grid with the respective values. For example: CIAO takes the current state of the simulation and passes it over to the ML-Interface. ML-Interface will then go on and hand these values over to AIxeleratorService, which will then perform the inference on the GPU. . . . 25
4.2. The architecture of the ML-Interface library[54]. 27

4.3. The distribution strategy from the AIxeleratorService library. [54] `RoundRobinDistribution` implements a correct but simple algorithm for assignment while `gpuAffineRRDistribution` [30] performs an optimized algorithm that also accounts for the layout of NUMA nodes.[39] 28
4.4. The communication strategy for data exchange across processes. [54] The library offers two implementations: `CollectiveCommunication` uses the entire communicator group in one step and `NonBlockingPtoPCommunication` that performs individual communication between processes. 29
4.5. The inference strategy with its implementations in AIxeleratorService [54]. 30
4.6. A complete overview of the AIxeleratorService library [54] with all its components. 31

5.2. Based on the previous code snippet, we added steps 6 and 7 which express the data streaming and parallel aspects. There is no longer a specified limited to how much data the training will consume and it will continue running as long as there is data available. The gradient exchange step (Step 7) is executed after computing the local loss and before applying the gradients. The numbering from the chapter 2 was kept. 35
5.3. Melissa Overview 36

6.1. Communicator Design 40
6.2. We extended the distribution strategy [54] 4.3 by another entry, the communicator for syncing gradients. The new member is colored red and the method we needed to modify green. `gpuAffineRRDistribution` [30] was not modified and does not create an additional communicator. . . 41
6.3. All communication strategies are now capable of copying the output samples in addition the input data. Additions are colored in red. 6.3 . 41
6.4. The newly proposed training strategy for AIxeleratorService. It provides an abstraction for selecting between multiple libraries such as Tensorflow, ONNX and Torch. In green we highlighted methods that we suggest but did not implement. 42
6.5. A complete overview of the AIxeleratorService library [54] with all its components. 43
6.6. The architecture of the ML-Interface library [54]. For clarity we have omitted types and functions used internally. *Italic* functions are abstract and implemented by the respective children. 44
6.7. Data extracted from AIxeleratorService using the described BMP[44] format. Bright regions correspond to large values for the progress variable and dark regions to lower values. 45
6.8. An abstract view of the training process. Function parameters are omitted for readability. 46

7.1. A correctly identified digit that was not part of the training dataset. . 47
7.2. MNIST training accuracy 48
7.3. A histogram showing the distribution of the variance of 2479 samples. 52
7.4. 53
7.5. Benchmarks when training our MNIST model on the GPU. 54
7.6. 55
7.7. Benchmarks when training the same UNet++ model on GPUs. . . . 56
7.8. The training loss of our UNet++ on simulation data after applying variance based filtering. 56

A.1. Benchmarks when training our MNIST model on the CPU. 62
A.2. Benchmarks when training our UNet++ model on the CPU. 62
A.3. 63
A.4. 64

List of Tables

A.1. Specs for a single node with one GPU 61

Declaration on LLM Usage

The application of LLMs in this document is limited to figures and visualizations. LLMs have been used to generate or modify images that are based on the `TikZ` library and to assist organizing the bibliography. They were **not** used for text writing, `matplotlib`-based visualizations and the development of core functionalities for the software this thesis is about. LLMs have been used to assist the translation of Tensorflow based models to PyTorch and are cited with the corresponding model given in a seperate bibliography at the end of this document.

All generated output has been personally verified and any remaining inaccuracies are to be considered my own failure.

1. Introduction

1.1. The Limits of Model Training

When simple Neural Networks started solving *real-world* problems years ago, they were given tasks like deciphering hand-written digits [40] or spam recognition [17]. Solving these tasks did not require a complex network structure [60] nor did it need a lot of training data when compared to today's standard [28, 77, 40, 45]. It was usually the case that all training data fit into the system's main memory and only had to be loaded once during startup.[40, 45] For the case of hand-written digits, the MNIST dataset was a popular choice and its overall size was less than 15 MBytes. With time, models grew in complexity and so did their need for more data.[45, 63] In 2006, a dataset for text processing released by Google already contained one trillion words.[28] The filesize of that corpus was 24 GBytes when compressed and it was already apparent that the demand for data would outgrow the capabilities of computers in regard to storing all that data.[28, 49] At the same time, *Support Vector Machines (SVMs)* [16] became popular for answering simple, interactive question based on the training data and showed the immense potential of learning algorithms when combined with enough data. [28]

In the 2010s, machine learning transitioned to GPU based workloads with one of the first models, that used GPU accelerated training, being AlexNet [37]. Krizhevsky et al. used 1.2 million images for training and stated that one of the main limiting factors was the available memory. In order to be able to load all the data, they split the training process onto two devices with each device only handling parts of the data and model. Their effort was supported by the fact, that they did not need to train the entire ImageNet [20] dataset but only a fraction of it.

Another example is the steady increase of data used for training the GPT models by OpenAI [77]. The first version of GPT was trained using 4.8 GBytes of data. Its successor, GPT-2 used almost tenfold of that with a total of 40 GBytes. This trend continued to GPT-3 that was trained on 570 GBytes. [77]

The overall trend is clear, machine learning applications will continue to consume more and more data - in volumes that do no longer fit on a single machine. For this reason distributed learning approaches have been developed that split the training process across multiple machines.[18, 37, 49] But even these techniques are limited, and while it is possible to increase the computational power by adding more GPUs to the training pool, the limited imposed by I/O bandwith will soon limit our computational power.

On the CLAIX-2023 cluster [67], the data rate peaks at 391 MBytes/s for disk

I/O. Training a model, that requires TBytes could already lead to significant delays and an underutilized set of GPUs. One example, besides GPT, that could already experience such behaviour are data-driven models working on simulated combustion data. [21, 54] These models are based on training data that is generated using highly concurrent simulations [21]. Due to its high dimensionality and volume, this data tends to have high storage and bandwith needs. [49] For this reason, new solutions are required that prevent the usage of expensive disk I/O in the first place. Melissa-DL [49] has already shown that training using live simulation data is possible, while also maintaining the model quality of offline training. In this thesis, we want to explore online and distributed training techniques for data-driven models designed for H_2 combustion processes. These models are trained using data from the simulation software CIAO[21], which was, so far, done offline using traditional training methods. It is our goal, to eliminate the need for disk accesses when handling CIAO simulation data and implement a functional training procedure within the AIxeleratorService[54] library that has so far been used for model inference within CIAO.

1.2. Outline & Contributions

In the next chapter, we first cover the data sources we will consider for training as well as the models designed for them. We will conduct initial tests that show that these models are indeed capable of learning the structure of the data. We will then move on to describe the overall training process that is used everytime a models is trained. In this context, we will cover how gradients are computed and applied to generate a model that is capable of generalizing with regard to its inputs. Afterwards we will outline the software and libraries we will use for our implementation of online learning. We will extend our previously established methods by ways to perform online and distributed learning steps and explain how this can be integrated. Finally, we will conduct an evaluation based on our training implementation in regard to model performance, meaning the quality of the output produced by the model, and computational performance. The contributions of this work are as follows:

- Outline how distributed and online training works related to traditional approaches.
- Coupling online training with the simulation task described before. We provide a concept for the AIxeleratorService library and the corresponding implementation.
- Verification of the presented implementation using simple methods like hashing and visualizations.
- A filtering algorithm that allows us to reject samples that do not provide a meaningful improvement for our model during training.

- An evaluation regarding model quality and training performance and outlook on future improvements.

2. Real World Data

2.1. Chemical Simulations

When working with gases, one usually starts by introducing a model based on the the *ideal gas law*.[65]The law itself relates the pressure p (in Pa) and volume V (in m^3) to the temperature T (in K):

$$p \cdot V = n \cdot R \cdot T$$

The amount of substance n (in mol) is usually assumed to be constant. $R = N_A \cdot k_B \approx 8.315 \mathrm{J\,mol^{-1}\,K}$ [65] is the product of the Boltzmann constant k_B and the Avogadro constant N_A. The latter defines the number of elementary entities within an amount of substance. Its value $N_A = 6.022\,140\,76 \times 10^{23}\,\mathrm{mol}^{-1}$ allows us to make an initial guess with how many molecules we are working. For example, 1 mol of water weights 0.018 kg[2]. It therefore does not take a lot of water to surpass 1 mol.[1] This will lead to a multiple of N_A number of entities in our process. It is clear that we do not have the computational power for simulating matter with such detail and are bound to use more abstract models such as the transport equation for transport and chemical transformation[59, 38, 75, 73]:

$$\frac{\partial}{\partial t}\rho Y_i + \nabla \cdot (\rho \mathbf{u} Y_i) = \nabla \cdot (\rho D_i \nabla Y_i) + \dot{\omega}_i$$

with

- Fluid density ρ
- Mass fraction Y_i
- Time t
- Mass diffusion coefficient D_i
- Velocity field $\mathbf{u}$
- Reaction rate $\dot{\omega}_i$

As presented in[59, 73] this model can be simplified further by focussing on the progress of the chemical reaction instead of handling multiple equation for transport of mass. The progress is stored in a multidimensional field called the *progress variable*[27] and each entry is normalized to be in between 0 (no reaction has taken place) and 1 (the reaction is complete). For a progress variable C we then get the following equation[61]:

$$\frac{\partial}{\partial t}\rho C + \nabla \cdot (\rho \mathbf{u} C) = \nabla \cdot (\rho D_C \nabla C) + \dot{\omega}_C$$

[1] One liter of water contains about 55.56 mol

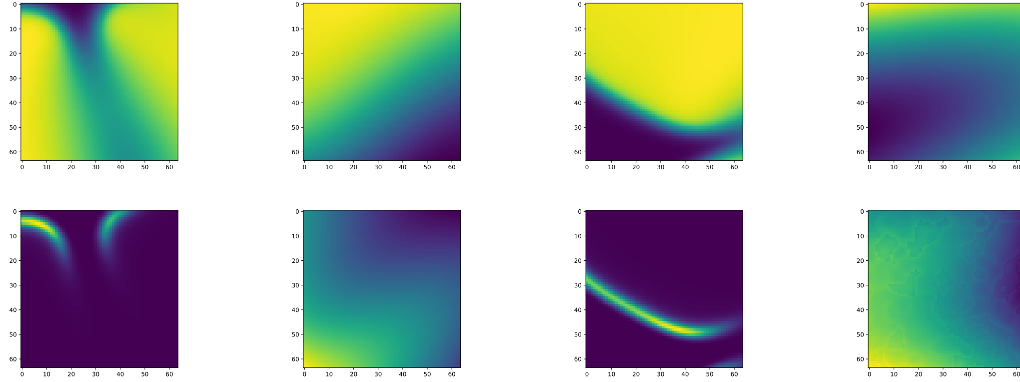


Figure 2.1.: Examples for data that is produced by a simulation.[21] The colors indicate the H_2O mass fraction with bright colors representing more H_2O and darker colors less. All samples have the dimensions `64x64`.

The new variable D_C represents the diffusivity of all involved species but the fuel.[61, 73] Just like Vanvinckenroye, we are covering hydrogen combustion processes in this thesis. This makes hydrogen our fuel, oxygen our oxidizer and H_2O our final product. We will also adapt the H_2O mass fraction as our progress variable.[73] At the beginning there will be no water present (our progress variable is 0) and during the reaction, the hydrogen atom will bind with oxygen to form H_2O molecules. The H_2O mass fraction is for that reason expected to only increase over time.

2.1.1. Simulating using Neural Networks

Because one single step within the simulation takes considerable amount of time7, it is desirable to find solutions that run faster. One such solution could be to employ a *neural network* to predict the next timestep.[49, 54] While the solver we use, CIAO[21], runs exclusively on the CPU, a neural network could be queried using a GPU instead. Depending on the size of the model, it could also be justified to theorize that CPU- based inference can be faster than simulating each step using a mathematical model. On the contrary, a small model could also reduce the quality of each inference step and for this reason, to prevent preliminary downscaling, we want to keep the model on the GPU for now. The UNet++[54] model we want to use is based on a UNet[64] design and is presented in [64, 54]. A UNet uses *convolutional* layers[55] to scale down the input by a factor over multiple steps. For example: The input might be of dimension `2048x2048` and after the first iteration is shrunk to `1024x1024` and after another iteration is further reduced to `64x64`, and so forth. After a UNet reaches the lowest level, it will start a process of scaling back up, where all these steps are taken in reverse and we will see an increase in dimensionality again. This is usually done until we reach our initial dimensions again, `2048x2048` in our example here. Of course, the downscaling cannot be done

arbirarily and only within reason. If we scale down too fast, we might run into issues that the resulting data lost some information and we will have issues scaling back up again.[24] On the contrary, taking took many steps for downscaling would cause an effect called vanishing gradients.[56] In this case, our model has grown too big and the weight updates are no longer reaching all parts of the model. This would then cause a slowed down training process, sometimes making it impossible to train. To combat this effect from happening, skip connections[76, 56] are introduced in the model. They are connecting different parts of the model that operate on the same dimension. In 2.3 these skip connections are colored in teal[2].

In previous works, it has been shown by Orland et al., that the UNet++ model is capable of learning from data that is generated from combustion processes.[54] The model was trained initially in an offline fashion and then performed inference on new data to aid simulation throughput.[54] In this thesis, the same UNet++ model will be used for training.

The data is collected from a simulation over an extended period of time and stored to disk. After the simulation is finished, the model is then applied on said data one by one. Here, the data is then read from disk again and temporarily stored in main memory. If the total amount of data exceeds our main memory, we need to re-read it from disk again after each epoch. Assuming that we have a field dimension of `2048x2048` and store each value as a 32bit float, each field would amount to 16 MByte of data. And storing 1000 samples would lead to about 16 GByte of data. However, this would then require the model to converge and generalize on only 1000 data samples. For complex models, this would be a sign of *overfitting* [68], e.g. the model would simply remember the input data and produce incorrect results for new samples. Training a model for proper results would require to use more than just 1000 samples and this could very easily lead to over 1 TByte of data.

Although we stated that the input data for our model will have the dimension `2048x2048`, this is not fully correct. This is the dimension on which the simulation operates, but the simulation is split into multiple parts. [21] The number depends on the total number of threads launched. While the overall field has the dimensions `2048x2048`, this field is split among all threads and each thread is assigned a local area that it needs to simulate. These areas are then connected using boundary conditions to guarantee that the entire field is sound. We decide to split the field into `64x64` blocks and process these in our model. In total we end up with $\frac{2048^2}{64^2} = 1024$ fields with reduced dimensionality across all threads. Training on simulation data from H_2 combustion data is already possible in an offline scenario, as stated above. We would still like to to get a first impression of training behaviour for later comparisons. For this we will use a reduced dataset of 5000 samples. These samples are trained over 50 epochs and split into a test and train dataset. When evaluating the test dataset we get results presented in 2.4. We can clearly see that our model is capable of learning structure within the data. This initial test is also neccessary as our implementation, later in this thesis, will be based on PyTorch instead of

[2] blue-green

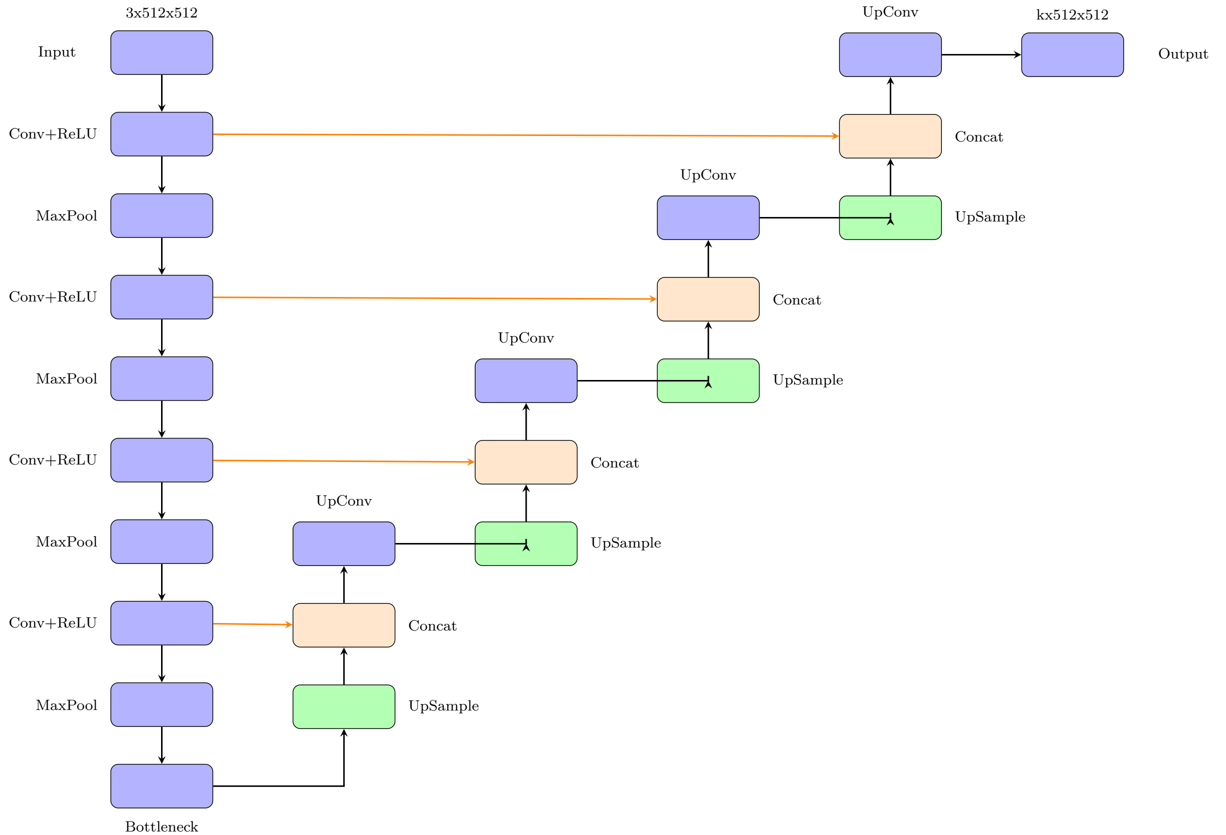


Figure 2.2.: The original UNet architecture
[64].[82]

Tensorflow as it was used in [54]. We fully translated the model into PyTorch with minor assistance from GPT-5[53]. Following our adjustments to the model, we conducted one training run with 20000 samples and 200 epochs. This result is given in 2.4. The training process generated many data points: $\frac{20000 \cdot 200}{800} = 5000$. On the logarithmic scale, this data has a high variance which ruins every visualization attempt. For this reason the trainig loss in 2.4 is plotting after applying a moving average onto the data. The windows size select here is 10. We can see that the increased amount of data and epochs results in a different convergence of the loss when compared to the previous attempt. We achieved results using the test set close to 1×10^{-4} while our attempt with less data only got near 1×10^{-3}. This motivates a deeper look into further experiments with more data and more epochs.

2.2. MNIST

Although our focus lies on simulations of hydrogen combustion processes, we also want to introduce the MNIST [40] dataset by LeCun. The dataset is a collection of handwritten digits coupled with information about the actual value. Since its

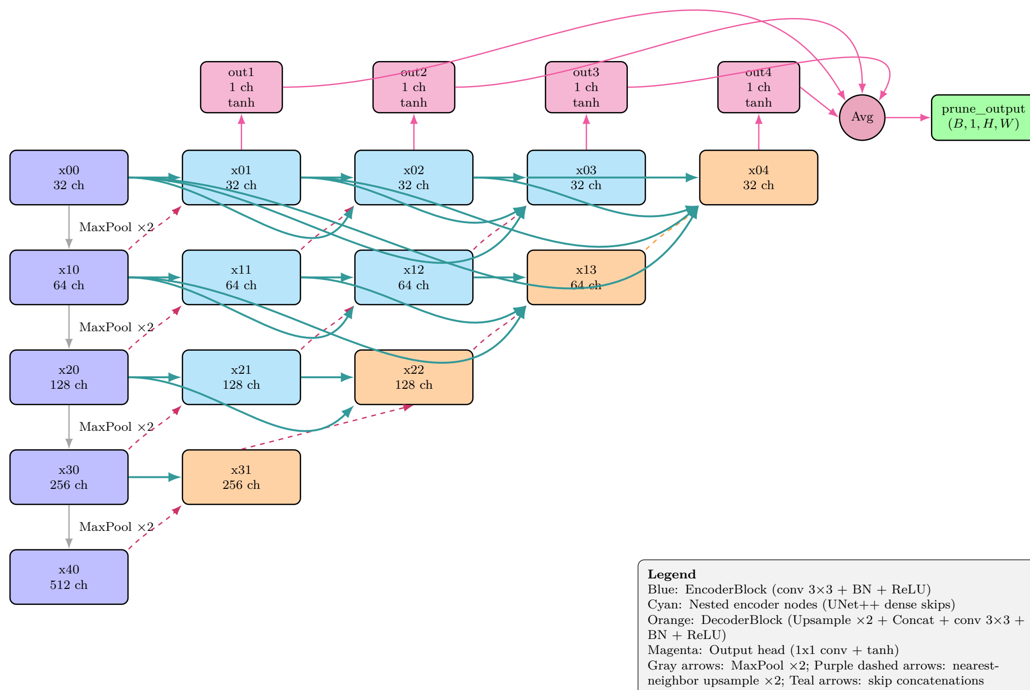


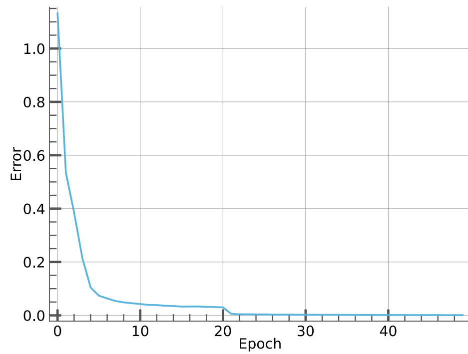
Figure 2.3.: The model UNet++ used for training on chemical simulation data. The model starts at x00 at the top left corner and ends at prune_output at the top right.[54][83]

publication in 1994, many models have been presented that solve[3] MNIST.[**?**] It is therefore not of any particular interest to us to present our own model and we will use an existing one that is already proven to solve MNIST reasonably well. [1]

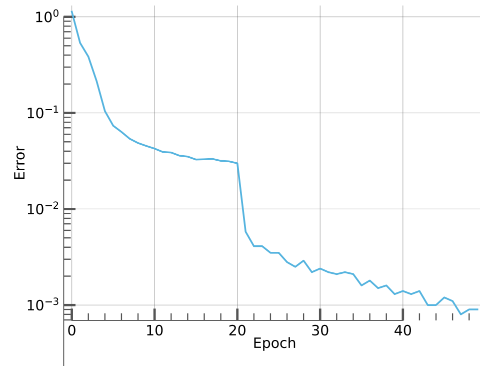
It is now a valid question to ask why we would even present MNIST in this thesis as its size is so small that it could easily fit into most computers memory or VRAM. Our initial setting was about data that does not have this property and needs to be loaded continuously during runtime. While it is true that MNIST does not have these requirements, it serves as a proof of concept for our application. We have the ambition to present a working solution to the previously presented problem and such a solution can be evaluated with more confidence when covering a problem that is already fully understood.[4] There are also alternative datasets such as EMNIST *(Extended MNIST)*[14] which extends MNIST by upper and lower case letters. It keeps the same 28×28 input dimension and could be used as a simple drop-in

[3]We say a problem or dataset is solved, if there exists a model that can learn the dataset with a high accuracy. For MNIST it is almost trivial to get an accuracy above 99%.

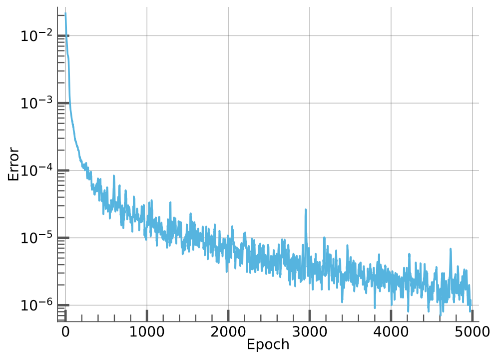
[4]MNIST will not have any impact on our data handling regarding the chemical simulation. It is only used to verify and present our implementation. Subtle bugs are quickly overlooked when working with new data.



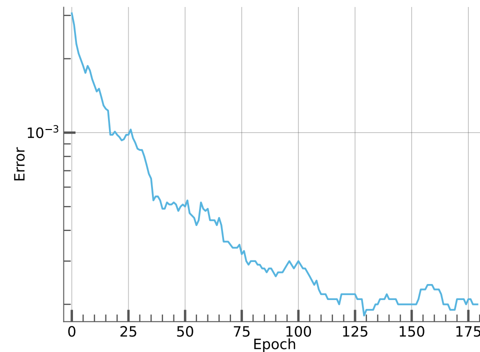
(a) Linear Error



(b) Logarithmic Error



(c) Logarithmic Training Loss



(d) Logarithmic Test Loss

Figure 2.4.: The results of our first experiments with the UNet++ on simulated combustion data from CIAO. The model was trained on 50 epochs in figures (a) and (b) and on 200 epochs in figures (c) and (d). The training process utilizes 20000 samples.

replacement.

For our purposes, MNIST will be enough.

2.3. A Model for MNIST

We already pointed out that MNIST has been solved with low error rates in the past. One such case is MCDNN [12] which was published by Cireşan et al. in 2012. It achieved an error rate of 0.23% on MNIST and showed a considerable improvement compared to their previous model from 2010.[13] Back then they used an MLP approach to predict the labels in MNIST and achieved an significantly higher error rate of 0.35%. With MCDNN, Cireşan et al. developed a model that combines multiple CNNs into one final result. As they have shown in [13] and [12], CNNs are better suited for vision tasks than classic MLPs. For this reason we will

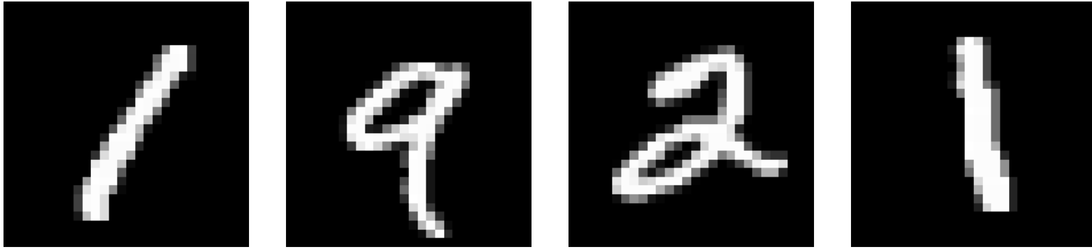


Figure 2.5.: A visual representation of the MNIST dataset. [40]The pictures were directly created from the dataset itself. Each pixel is encoded as a floating point value that represents its brightness.

employ a CNN based model for our use case.

PyTorch [57] is a library for neural networks on computer hardware which supports inference and training on CPU and on GPU via CUDA. The core library is written in C++ but is also made available to users using Python through a high-level API. Since the library is commonly used in an educational context, it provides many examples on how to use it including an example to train on MNIST.[1] The model published by PyTorch combines two CNN layers with a `ReLU` activation function in between and a pooling layer at the end. After that it acts similar to a MLP by using two fully connected layers. The result is then computed using a softmax function.

In 2.6 we provide a visualization of the model presented by PyTorch[1]. The model starts at the top and works its way towards the bottom. The convolutional layers *(Conv2d(. . . , . . .))* are seperated using a non-linear activation function. *Dropout(. . .)* is surrounded by a dashed lined since it is only active during training. Afterwards these layers are disabled and only forward their inputs. The second part of the model represents two coupled MLPs. At first, we transform our vector into a one dimensional representation and then pass it to the first part which is a fully connected layer *(Linear(. . . , . . .))* and then apply another activation function. Then we use a second fully connected layer to produce a result that contains exactly ten entries. The predicted category will have the highest value among all entries inside the vector and it can be amplified using a *Softmax* layer.[5]

2.4. Training the Model on MNIST

We will now train the model on the MNIST dataset to show that it converges. In doing that we will find that it converges very fast and we only need a tiny amount of epochs. As we have seen before, training a CNN model on MNIST is not a new discovery. For us this step is primarily intendend to verify that the model we chose [1] is appropriate and that the implementation is working. Furthermore we will use the results we obtain here for comparison with our implementation later on.

[5]Softmax also has the very desirable property of being a differentiable function which is important for training the model but that is not important for us at this moment.

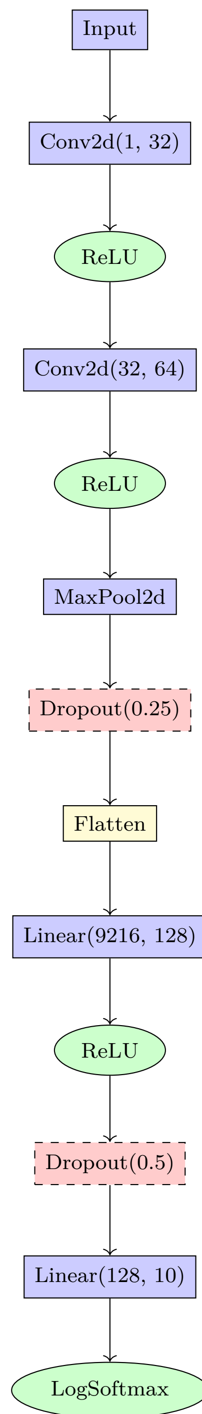


Figure 2.6.: A visual representation of the model we train on MNIST. [57, ?][78]

The model was trained on the dataset a total of $n = 30$ times, each time with a total of 14 epochs.[6] MNIST itself is split into a *train set* and a *test set*. The test set contains images that are not included in the train set, so we can evaluate the model on data it has not seen before. This is done to detect a common problem in machine learning called *overfitting* [68]. When overfitting occurs, the model is no longer learning the structure of the input data but only the correct output for each input. This makes the model mostly useless for working on new data. During each epoch, the entire train set is applied to the model and in between epochs we calculate the loss on the test set. One such run is shown in 2.7. We can clearly see that our model keeps improving during training and that the limit on epochs is justified. Furthermore, we observe that the loss already starts very low after the first epoch. The reason for this is that our model already updates its parameters during the first epoch and therefore is able to score a satisfactory result on the first test. For completeness we include the loss of the training set in 2.7. We used a batchsize of $b = 64$ during training, which means that the entire training process is split into

$$\lceil \frac{\#\text{Samples}}{\text{Batchsize}} \rceil = \lceil \frac{60000}{64} \rceil = \lceil 937.5 \rceil = 938$$

iterations. For a total of 14 epochs, this means a overall number of $14 \cdot 938 = 13132$ iterations throughout the entire training process. In the figure, we can note that the loss starts much higher during training, at a value beyond 2.0, but then drops quickly during the first epoch. For each of our 30 attempts at training MNIST we achieved one final evaluation against the test set after the last epoch. This value can generally be considered to be the best result of the model across all epochs.[7] We collect these results for each run and present them in 2.8. In the plot it is apparent that we achieved a very low variance in our data. It is therefore expectable that our implementation achieves a very similar value when training MNIST. We will come back to this claim to see if we can hold up to this.

Towards the end of this thesis we will also conduct a performance benchmark and analysis of our implementation. We will use this chapter for a first insight in how fast we can train a basic model on MNIST. In 2.8 we show the range of our measurements, including the mean value of 94.54 seconds. Using our previous considerations regarding the total number of samples and the used batchsize, we get a runtime of $\frac{94.54\text{s}}{13132} = 7.1992\text{ms}$ per batch. The processing of a single batch includes *1)* a forward pass through the model *2)* computation of loss in regard to the expected value *3)* a backward pass and finally *4)* an invokation of the optimizer. We will only mention these steps here and move a more detailed evaluation to a later chapter.

This concludes our chapter on MNIST and our data sources in general. We have seen that MNIST is an easy problem to train models on and that we can use it for analysis of the implementation we build for more advanced models. We have

[6] This number is the default setting in the script provided by PyTorch. However it is also a reasonable setting as we are not getting any meaningful improvement after 14 epochs.

[7] At least the value will be very close to the actual minimum since progress slows down after 10 epochs and the loss starts fluctuating.

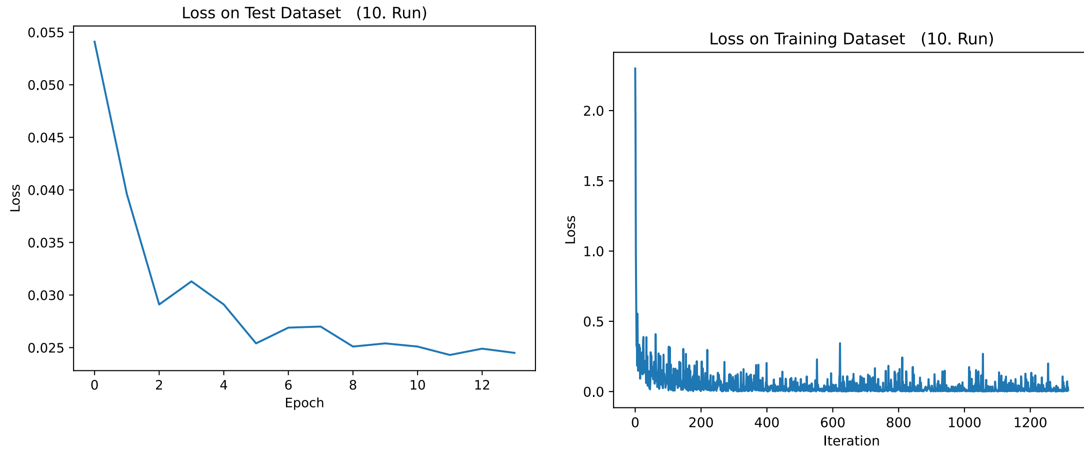


Figure 2.7.: The tenth run training the model on MNIST. It is representative for all iterations we did when training MNIST. The loss is computed on the test dataset after each epoch. It is clearly visible that we get a significant improvement throughout the entire training process. The loss we get on the training set during our entire run. We can see that the loss is starting much higher than on the test set.

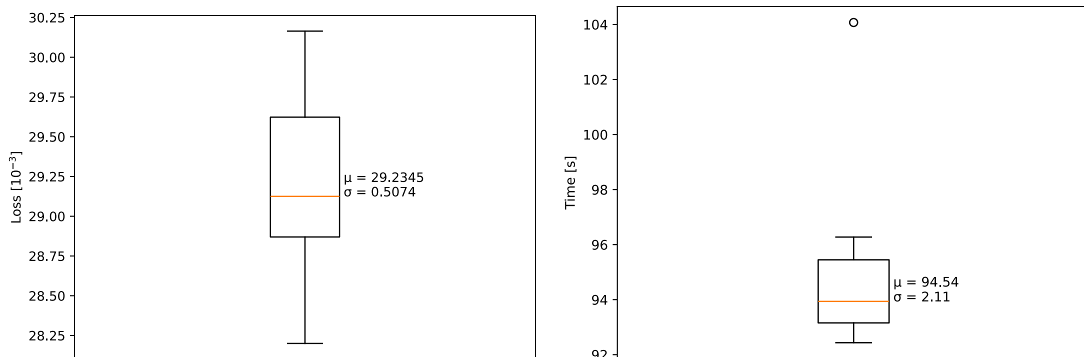


Figure 2.8.: The loss across all 30 runs with a mean value of around 0.0292345 Note the scaling on the y-axis. The runtime across all 30 runs with a mean value of 94.54 seconds.

introduced the MNIST dataset and a model that can be trained on this model successfully. In addition to that we have introduced the data we receive from our reactive flow simulation and the UNet++ model, that can be used for training on this data. We verified the capabilities of this model as well by training it on a pre-generated dataset. While were only a user of a machine learning framework so far, in the next chapter, we will take a closer look on how this training process is really working so we can then take the next step and implement training based on streaming simulation data.

3. Training a Model

In the previous chapter we have briefly touched to subject on how a model is trained. More concrete: We have just used the infrastructure provided by PyTorch to train a model and the reason for that was that we only wanted to show the models are capable of producing correct outputs. To understand distributed training, we need to establish a foundation on how models are trained. The setting will be the training of a model with just one singular GPU. At first, we we will discuss how training works in this scenario and then proceed with distributed and online training.

3.1. How to Train Your Model

When training a Model, its performance is measured after each forward pass against an expected value.[8] A model can then be viewed as a function $m(\cdot)$ that maps every input I to an output $O = m(I)$. This output is then expected to be close to some given ground truth G. Machine learning with this pattern is called supervised learning[8] and is the only type of machine learning we will be discussing here. Hence, we will always expect to be given pairs (I, G) that we can feed into the model. While this encompasses the data for our model, we also need a way of comparing the model output with the ground truth. This is typically done using a loss function.[8] These functions compare the output with the expected value and return a scalar to indicate how close the output is to what was expected. PyTorch offers multiple loss functions out of the box, the most commonly used ones are presented in 3.1. These loss functions have different purposes, e.g. *CrossEntropyLoss* is used for classification and *MSELoss* for regression.[57, 35, 46] Overall we have the following components:

1. The input data $I \in K^{D_1}$
2. The output data $O \in K^{D_2}$
3. The ground truth $G \in K^{D_2}$
4. The loss function $l : K^{D_2} \times K^{D_2} \to \mathbb{R}$
5. The model $G : K^{D_1} \to K^{D_2}$

Given a model and some data, we are now interested in an optimization method that transforms a model in such a way that it improves based on seen data and generates outputs closer to the expected ones. This method consists of two ingredients:

Identifier in `torch.nn`	Short Description
`nn.L1Loss`	Measures the mean absolute error
`nn.MSELoss`	Measures the mean squared error
`nn.CrossEntropyLoss`	Measures the cross entropy between input and target
`nn.NLLLoss`	Measures the negative log likelihood loss

Figure 3.1.: Some commonly used losses available in PyTorch[57]

1) Backpropagation: An algorithm that calculates the impact each weight inside the model has on the final output[66, 8] and *2) Gradient descent optimization:* A method that will use this information to steer the model into the desired direction.[9, 8] We will now briefly cover them both and later build on that to bring distributed learning into the mix.

3.1.1. Gradient Descent Methods

The error $E(\cdot, \cdot)$ is given via the chosen loss function l.[8] Assuming a sum of squares error, we get the following result[8]:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \tag{3.1}$$

$$= \frac{1}{2} \sum_{n_1}^{N} \{m(I_n, \mathbf{w}) - G_n\}^2 \tag{3.2}$$

We now want to optimize the parameters $\mathbf{w}$ in such a way that $E(\mathbf{w})$ approaches some (local) minimum at $\mathbf{w}_0$ such that

$$\nabla E(\mathbf{w}_0) = 0 \tag{3.3}$$

For some machine learning applications, this optimization problem can be solved analytically[8]. Unfortunately, for all most interesting and relevant applications, this is not the case.[8] Hence, we won't follow this path here. An alternative to this are algorithms that evaluate the model and its loss function multiple times and then use the result as a clue to find **w0** . Relying solely on the input and output here would make this a computationally expensive procedure: In [8] Bishop presented how this approach would require $O(W^3)$ steps - where W is the dimensionality of $\mathbf{w}$. It is furthermore shown that gradient information can bring this down to $O(W^2)$ steps. A significant improvement when working with larger models and the main motivation for gradient descent. In we will see how these gradients can be determined - for now, we will assume that we already have a method for computing them. As pointed out in [8, 9, 41] optimizations for $E(\mathbf{w})$ are done by making minor adjustments to $\mathbf{w}$ in the direction given by $-\nabla E(\mathbf{w})$. Without gradient descent, our weights update looks as follows[8, 41]:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} + \Delta\mathbf{w}^{\tau}$$

Here, we would need to gues or estimate the value of $\Delta\mathbf{w}^{\tau}$.[1] With gradient descent, we know the direction of the closest (local) minimum. This can be used to improve the update step[8]:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} + \eta\nabla E_n(\mathbf{w}^{\tau})$$

In this formula, we make use of the information we have regarding direction. However it is still unclear how far we need to go in said direction. This step size or learning rate is encoded in the parameter η and is in practice considered a hyperparameter in machine learning.[8] Its value needs to be fine tuned for the model at hand and can not be calculated upfront. η is of utmost importance in machine learning as it has a tremendous effect on the time needed for training and the performance of the model afterwards. Picking a small η will result in a slow convergence process, but picking a η that is too big will result in overshooting local minima.[8] These models will not get the convergence results they could achieve. Therefore, when training large models, it is common to start with an initial value for η and decrease it during the training process.[8] The above provides an intuition for how the update step in gradient descent methods works. Once we are given some initial parameter $\mathbf{w}$, we can optimize it regarding some error function $E(\mathbf{w})$. Of course, this is only very fundamental optimization using gradients. Using this basic update rule, could already get us results. But we would not get the results modern optimizers would. These optimizers use additional techniques such as momentum to avoid getting stuck in saddle points or bad local minima.[8, 9, 36] For our purposes it is not required to delve into these topics and we can use these optimizers, like Adam[36] or SGD[9, 8]. The technical details of these solvers are not required for distributed learning and out of scope of this thesis. Until now, we have assumed that we just know the gradients $-\nabla E(\mathbf{w})$. To complete our tour on model training, we will now present an established method called backpropagation that is (partially) used for calculating gradients.

3.1.2. Backpropagation

Backpropagation[66, 8] is the technique for calculating the gradients after a forward pass. As we have seen before, these gradients are needed to identify the direction we need to shift w to reduce the error $E(w)$.[8] For each weight in our model, backpropagation yields the gradient based on the provided input data. Together these gradients will combine into the gradient $-\nabla E(w)$. The chain rule states:

$$\frac{dy}{dx} = \frac{dy}{dz}\frac{dz}{dx}$$

This can be applied to the error function $E(\mathbf{w})$ of neural network[8, 66]:

[1] As stated before, because $\mathbf{w}$ is a vector, this is very costly.

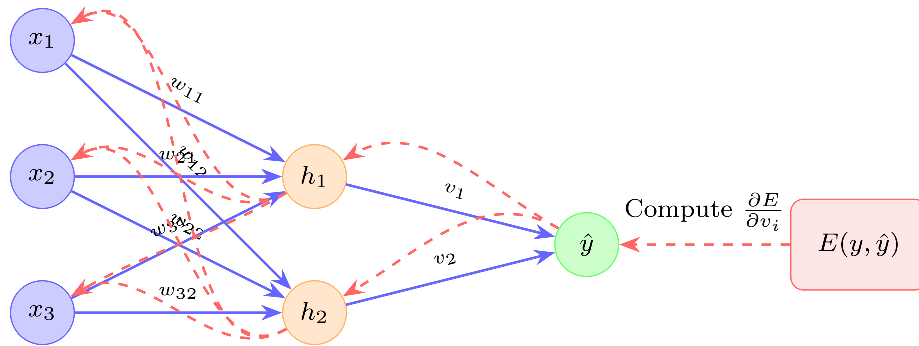


Figure 3.2.: Backpropagation[66] visualized for a feed forward network that uses one hidden layer (orange). The gradient is represented using red arrows tracing back towards the input nodes.[81]

$$\frac{\partial E}{\partial \mathbf{w}} = \frac{\partial E}{\partial z} \frac{\partial z}{\partial \mathbf{w}}$$

to get the change in error regarding every single operation within our network.

We have depicted a feed-forward network in 3.2, combined with a loss function at the end. The gradient of, for example, h_1 only depends on the input it receives from its immediate predecessors. For efficient training, now have to be able to compute every single gradient and its role in the loss value and the final result. For this, gradients are computed in reverse order, we are starting with $\hat{y}$, and then followed by h_1 and h_2. Using the *computational graph* 3.3, PyTorch applies *automatic differentiation* to calculate the gradients.[57, 8, 6] This avoids manual gradient calculations and allows for efficient training of complex models.

```
model = Model()
optimizer = Optimizer(model.parameters())
l = Loss()
for (I, GT) in data:
    optimizer.zero_grad()    # Step 1

    O = model(I)             # Step 2
    loss = l(O, GT)          # Step 3

    loss.backward()          # Step 4
    optimizer.step()         # Step 5

```

Figure 3.4.: PyTorch inspired sample code for training a model.

3.1.3. Basic Training Loop

After discussing backpropagation and gradient descent, we can now combine those and present a simple training loop for a single GPU. In 3.4, we have depicted the outline of the training process as far as we can trace it for now. We start by initializing the model, the optimizer and the loss function. In most cases the model is directly implemented in the same language we write our code in and can simply be instantiated. In Python and PyTorch[57] it is common practise to write the model as a class that inherits from `torch.nn.Module` and thereby possesses the neccessary members to be used like a common function. Such models can be instantiated like any other Python class and will automatically provide the internals for training and inference. Regarding the choise of the optimizer, PyTorch provides a generous amount 3.1. For us the only two important optimizers here are `torch.optim.SGD` and `torch.optim.Adam`. These are the two we will be using for training all our models. Data loading is done in batches for faster processing and gradient stabilization.[8] Inspecting the actual training loop, we can spot five distinct steps required for a full iteration:

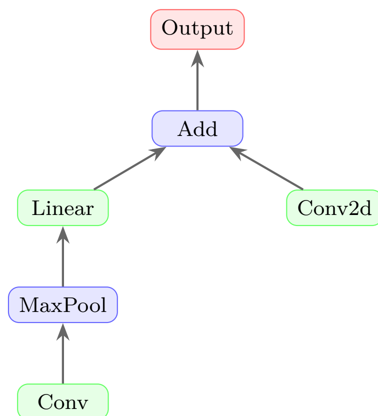


Figure 3.3.: PyTorch build an internal graph for each tensor that records all layers that were applied to it.[57, 6]

1. Resetting the gradients: Before we even pass our data through the model, we need to reset the optimizer. This is done using the `optimizer.zero_grad()` call. Because the optimizer is responsible for any changes made to the weights

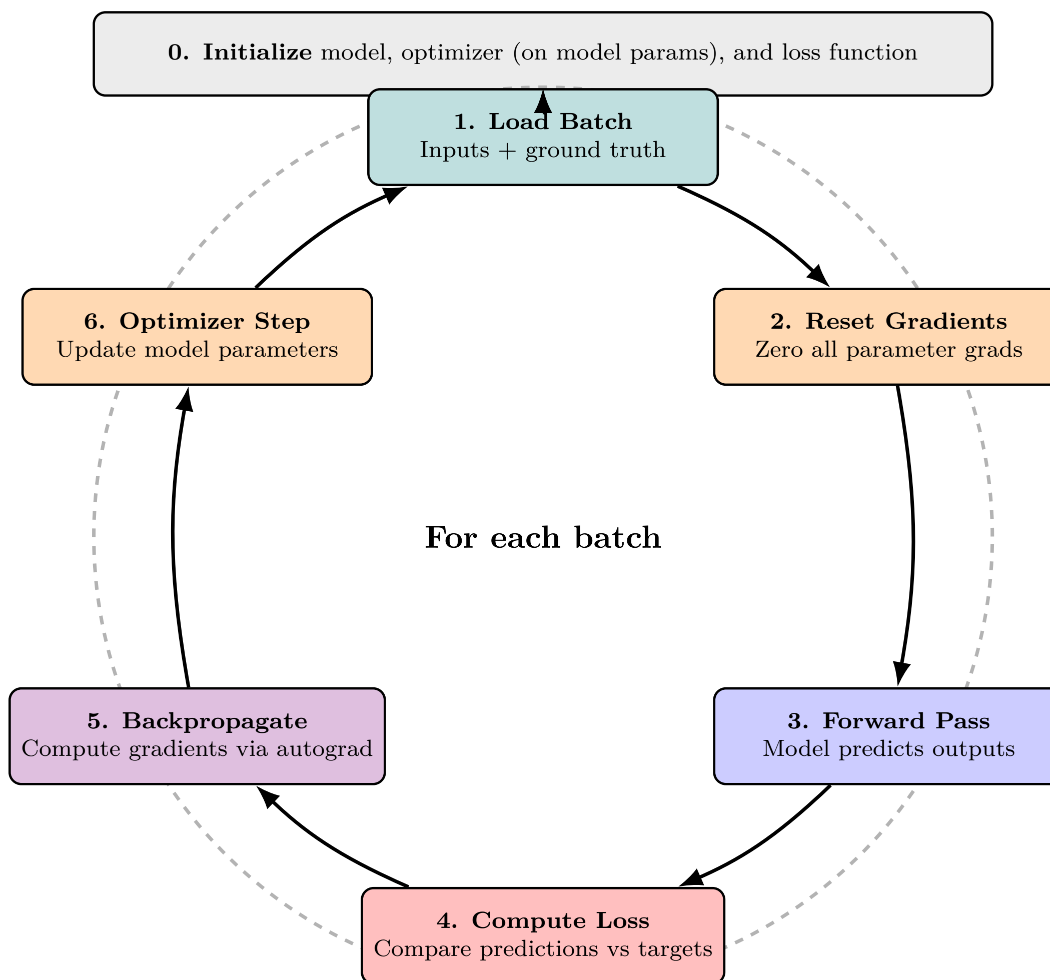


Figure 3.5.: A visualiation of the previous code snippet. The code runs as long as we have data to feed into the model.[84]

of the model during the training process, we need to set the internal gradients back to zero. This is to prevent any noise from the previous iteration to leak into our current step.

2. Forward pass: We then need to apply the model to our input data, so that we get an output value. In this step, PyTorch will not just evaluate the model for the given data but also bookmark which operations are used when the data is modified. This will be collecting in a *computational graph*.

3. Computing the loss: After we have evaluated the model for a given input I and produced an output O, we then compare the output with an expected ground truth G using the loss function.

4. Backward pass: Based on the result of the loss function and the bookkeeping during the forward pass, we can then calculate a gradient value for every weight

inside the model. The previously stored *computational graph* is then traversed backwards.

5. Gradient application: After we have calculated those gradients, we only need the optimizer to apply them on the model. By doing so, we hope to get an improved result in the next iteration.

We have now established the fundamentals for training a model. Our overall process here was highly influenced by how PyTorch is structured around training as this will be the library we will use for training later on. Other frameworks, like Tensorflow[5] and ONNX[22] realize a slightly different approach in regard to program structure, however the overall concepts remain the same.

Following our previous discussion, the following chapter would seem like a good place to start introducing distributed learning and learning with online data. But before we explore that field, we need to have a deeper understanding of the existing software besides PyTorch, that we will base our implementation on.

4. Enabling Technologies

4.1. An Overview

We will begin this chapter with a brief overview of the programs and libraries used as well as an outline of how these components are interacting. In this chapter, we will introduce each software as it was before this thesis. The changes we made will be discussed in a later chapter.

In 6.8 we visualize the different components and the API they use. The data here is usually passed by reference to avoid unneccessary copies. AIxeleratorService only expects a pointer and the corresponding sizes. From AIxeleratorService to the GPU we need to do a complete copy, since the data is only residing in main memory and we need to load it into VRAM.

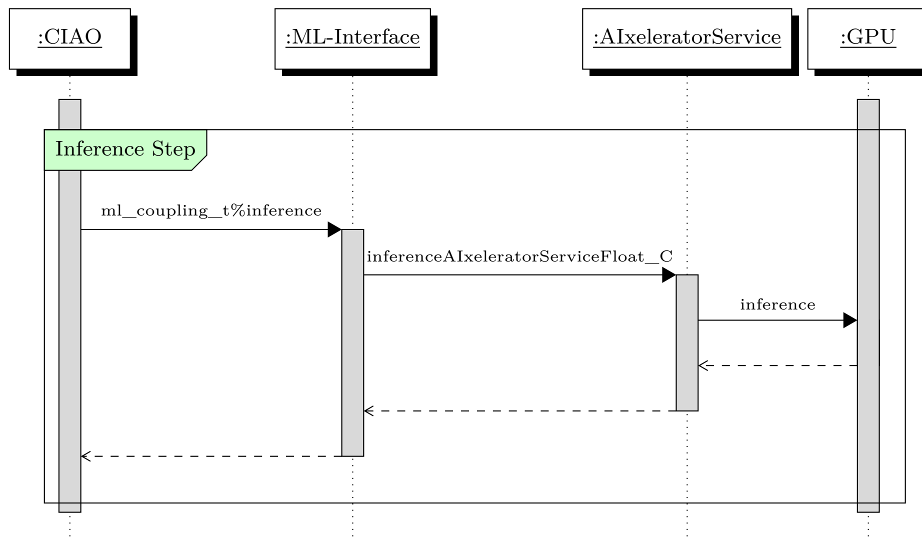


Figure 4.1.: An abstract view of the inference process. Function parameters are omitted for readability. In the inference case, each call usually passes the grid with the respective values. For example: CIAO takes the current state of the simulation and passes it over to the ML-Interface. ML-Interface will then go on and hand these values over to AIxeleratorService, which will then perform the inference on the GPU.

4.2. CIAO

CIAO[21] is the solver developed by the Institute for Combustion Technology[1] and will provide the training samples for the model. It supports *1)* Large-Eddy Simulations and *2)* Direct Numerical Simulations. [21] Furthermore, CIAO is highly parallelized and supports simulations that run across hundreds of processes on multiple nodes using MPI [48]. This makes CIAO an ideal choice as it is capable of providing new data quickly. We will use CIAO to simulate H_2O combustion processes.

When simulating, all processes are each assigned one subset of our `2048x2048` field. A larger number of processes results in smaller subsets, which increases the performance of the overall simulation. CIAO guarantees that boundary conditions between cells are met, using internal mechanisms, and that we are working with accurate data after each simulation step. Furthermore, CIAO provides us with a function that is called after each step by all processes. This call is synchronized among all processes such that no process can make progress beyond the current step while other processes are still computing. This function is a good entry point for the ML-interface 4.3 to copy the data from the simulation and start either an inference or a training step.

For the remainder of this thesis we will view CIAO as a black box that periodically provides us with new training data.

4.3. ML-Interface

The previously mentioned solver4.2 is almost exclusively written in Fortran [3]. In addition to that, the library we want to use4.4 is written in C++ [4]. ML-Interface[54] by Orland et al. is neccessary to enable seamless integration of both projects. It was designed to fulfill the following tasks:

1. *Translate column-major to row-major:* Fortran and C++ use different orders for indexing arrays. [3, 4] While Fortran uses column-major, C++ uses row-major. All code within CIAO and AIxeleratorService is designed to use the same array indexing, i.e. use the same order for indices. This requires a translation step within the ML-interface.

2. *Field (de)normalization:* A common technique for improving model performance is batch normalization. [34, 42] It enables the model to score the same accuracy but with 14 times fewer steps. [34] Generally, normalization ensures certain statistical properties of the training data while keeping the overall structure intact. [32] ML-Interface applies a normalization step before every call to AIxeleratorService and a denormalization step afterwards. [**?**]

3. *Stacking:* Batch training [42] is a variation of the training process where multiple samples are processed at the same time. When using batch training,

[1] `https://www.itv.rwth-aachen.de`

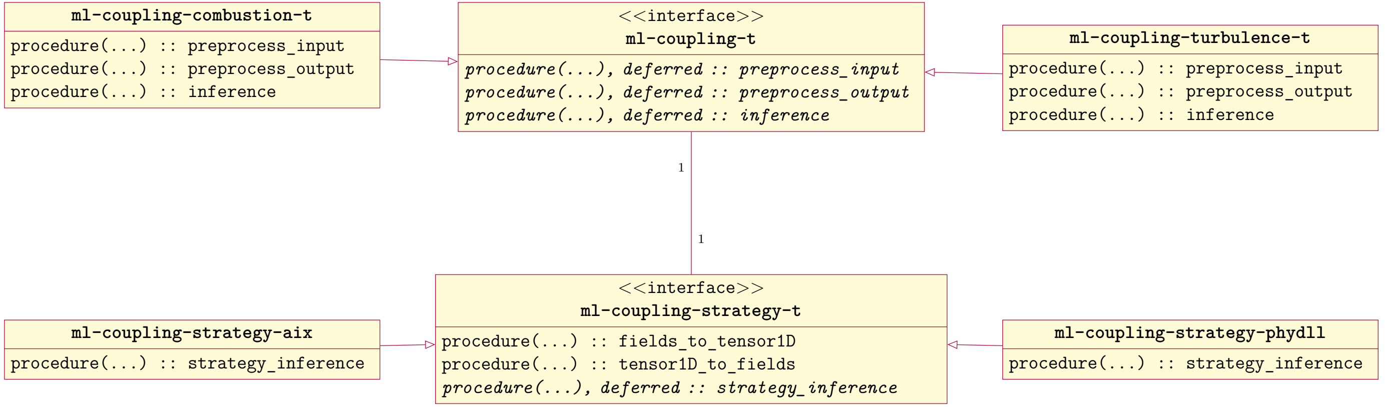


Figure 4.2.: For clarity we have omitted types and functions used internally. *Italic* functions are abstract and implemented by the respective children.

the dimensions of our input data changes and we prepend the batch dimension. For example, `64x64` becomes `4x64x64` when using 4 samples per batch. ML-Interface internally converts all samples into a batch with batch size 1. [54]

4. *Reshaping:* When passing data from Fortran to C++, we can only pass one dimensional data. [54, 4] For this reason we need to convert the arrays from one representation to another. ML-Interface handles this in between each AIxeleratorService call. [54]

ML-Interface provides two interfaces that together with their specializations make up the entire library. The inferfaces are used for selecting the simulation model and the inference backend to be used. [54] 4.2 For the simulation we can choose between a combustion and a turbulence process. Depending on the selection ML-Interface needs to perform different preprocessing steps before invoking AIxeleratorService. As a backend implementation it is possible to instaniate either the AIxeleratorService [54] library or PhyDLL [10]. This thesis only covers combustion models in combination with AIxeleratorService.

4.4. AIxeleratorService

The AIxeleratorService library [54, 73] by Orland et al. was developed for distributed inference on computer clusters. It is intended to be used in combination with other software instead of a standalone binary. For this purpose it provides the user with an interface for C, C++ and Fortran. Internally AIxeleratorService manages an MPI communicator for communication across nodes that is setup during initialization and provided by the embedding host application. Within the scope of this thesis, the host application will be the solver CIAO4.2 combined with ML-Interface[54, 21].

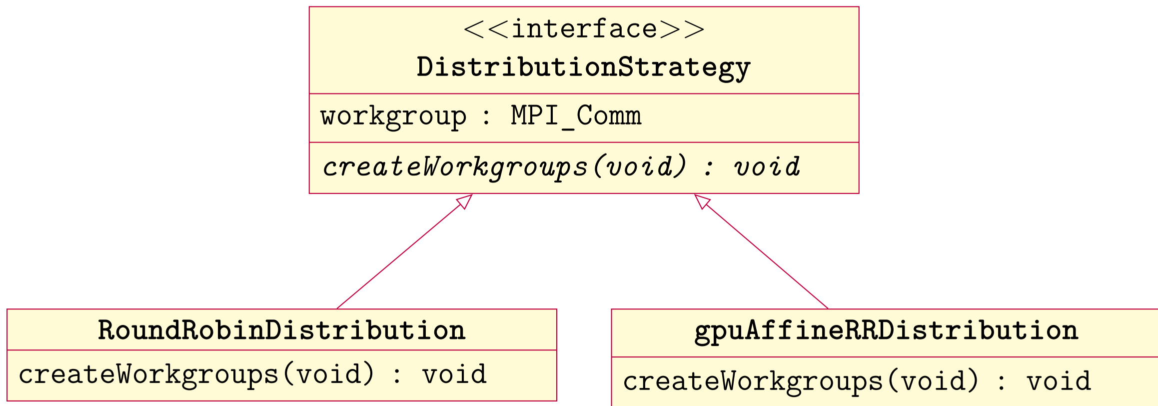


Figure 4.3.: The distribution strategy from the AIxeleratorService library. [54] `RoundRobinDistribution` implements a correct but simple algorithm for assignment while `gpuAffineRRDistribution` [30] performs an optimized algorithm that also accounts for the layout of NUMA nodes.[39]

The AIxeleratorService library is split into multiple segments that are designed after the *strategy*[26] design pattern. [54] This makes the library flexible to adjust and combine different methods and algorithms with various machine learning frameworks. During inference, each process has to initiate its own AIxeleratorService instance. These instances communicate with each other, at first, using the the distribution strategy, and second, via the communication strategy. The distribution strategy is only used during startup and assigns all processes into groups with each group containing at least one process controlling a GPU. 4.3 One specialization is the `RoundRobinDistribution`[54] that uses `MPI_Comm_split` to perform this assignment such that the GPU controlling process has the lowest rank within the newly created communicator. The next strategy within AIxeleratorService is the `CommunicationStrategy`6.3 which uses the previously created communicator for data exchange between processes. Since the number of processes with a GPU is much smaller than the overall number of processes, we need the `CommunicationStrategy` to handle the dataflow towards the GPU processes. We will use the `CollectiveCommunication` implementation for the `CommunicationStrategy`. AIxeleratorService itself does not invoke the GPU but implements different providers for established backends, such as Tensorflor [5] or Torch [57]. They make up the `InferenceStrategy` with its specializations. 6.4 In 4.6 we present an overview of the entire library.

We have now seen how the existing software stack is designed and operates. However, this has only been in regard of inference within the AIxeleratorService library, meaning that we copy the input data via ML-Interface to the AIxeleratorService and then perform a single inference step based on the simulated data. The resulting data from the model is then returned back to CIAO where it is used to perform the next simulation step. In chapter 6 we want to extend the architecture to enable training. For this purpose we need to pass the output data from the simulation

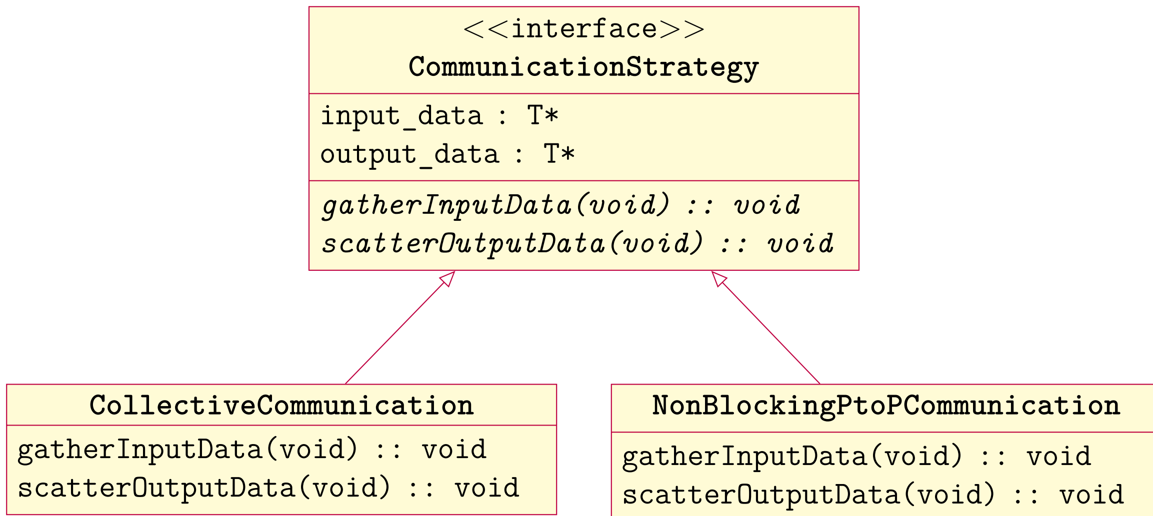


Figure 4.4.: The communication strategy for data exchange across processes. [54] The library offers two implementations: `CollectiveCommunication` uses the entire communicator group in one step and `NonBlockingPtoPCommunication` that performs individual communication between processes.

along with the input data to be able to perform a training step.

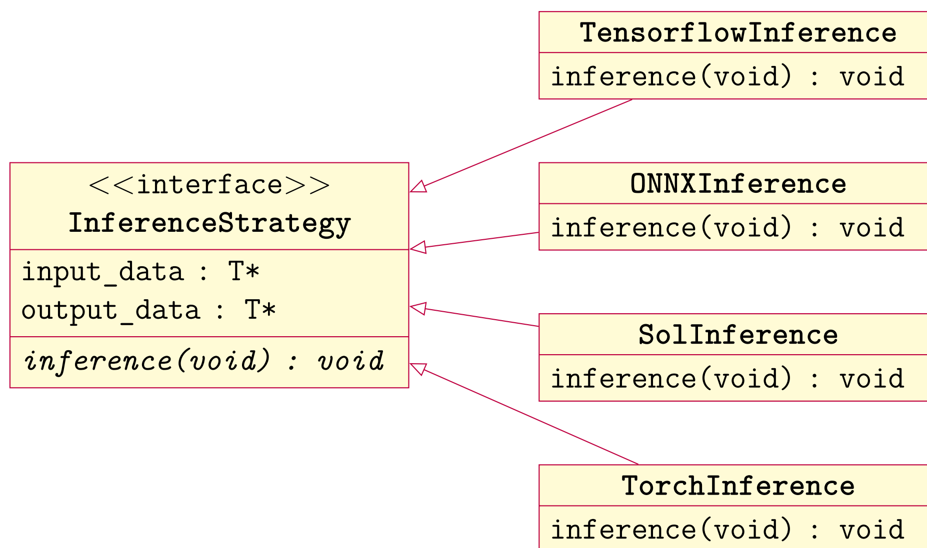


Figure 4.5.: The inference strategy with its implementations in AIxeleratorService [54].

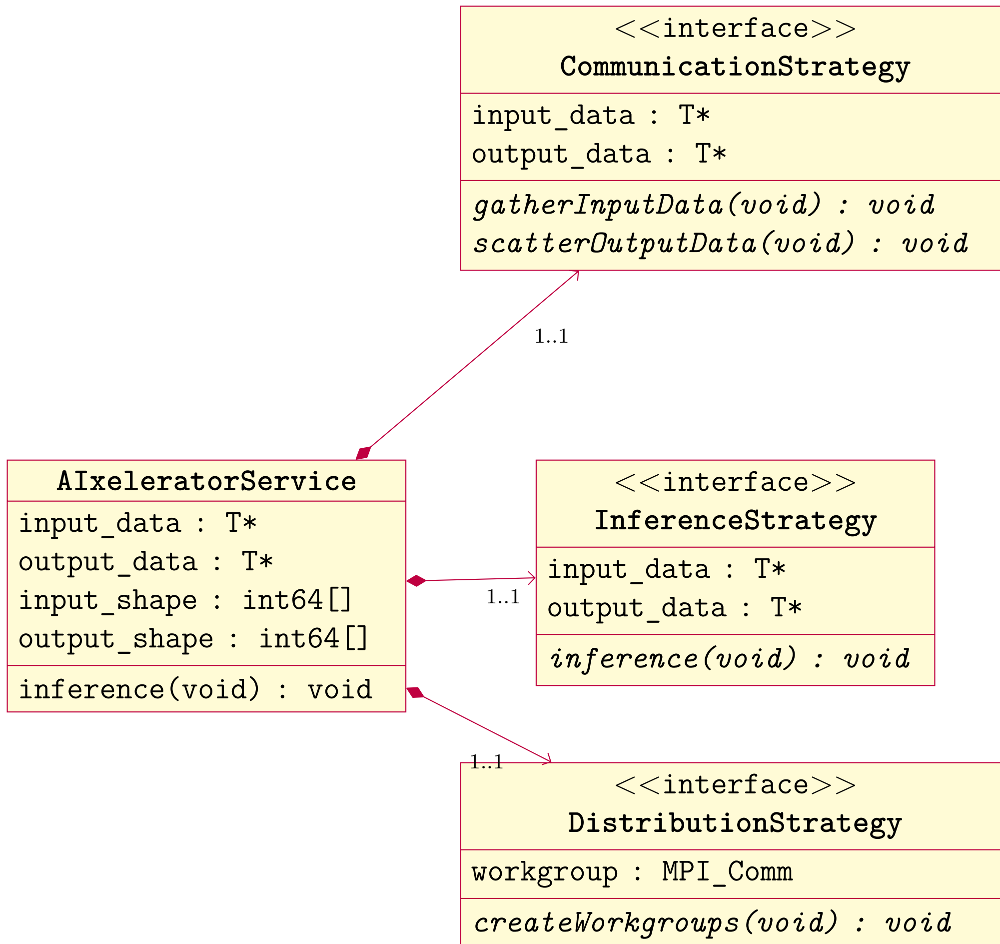


Figure 4.6.: A complete overview of the AIxeleratorService library [54] with all its components.

5. Distributed & Online

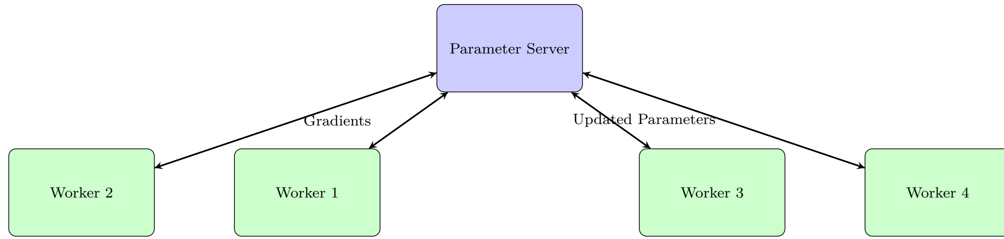
This chapter extends chapter 3 by concepts that allow us to train a model in a distributed fashion. We will then continue with online machine learning which - in this scenario we will no longer be required to read data from disk but instead are able to receive it via a network interface.

5.1. Training in a Distributed Setting

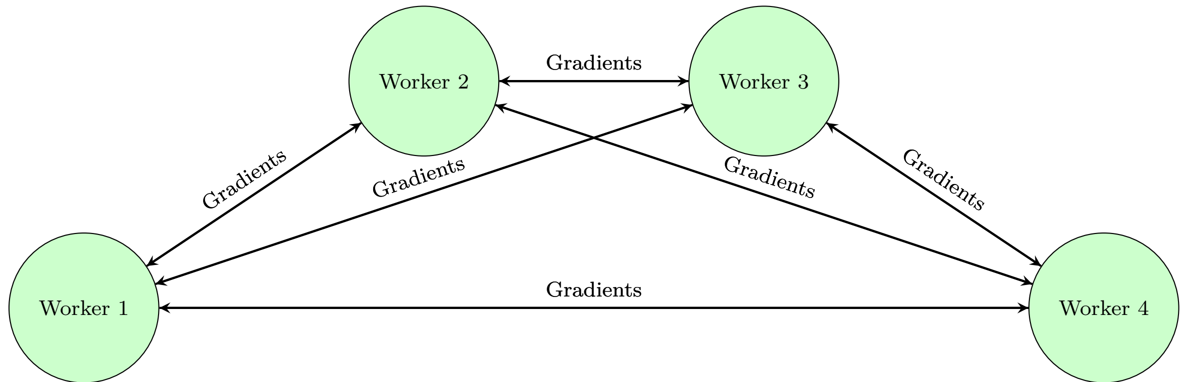
Training a model on a set of distributed nodes requires at least 2 processes running at the same time. We will refer to the total number of processes involved with training with n_t. Scaling the training process to n_t processes, one has to decide if the model and/or the data should be split across nodes. It is possible to run only parts of the model on some GPUs and other parts on the rest as it has been been done for AlexNet[37]. On the other hand, data parallelism could be the desired goal. Each of the n_t nodes initializes the same model architecture, but then received different parts of the data. The model is then trained, using that data, similarly to3. However, after each backward pass and before the optimizer step, the calculated gradients have to be exchanged with all other currently running processes. [47, 43, 19] Each process then uses these to calculate new gradients locally. A common approach here is to take the average gradient across all nodes. This is conceptually based on batch learning when training on a single node.

Regarding gradient exchange, there are two common approaches for communication. The first one is training using a **centralized parameter server**.[19] This method dedicates one of our n_t processes to the unique role of a parameter server. All remaining $n_t - 1$ processes start training their model locally and transmit their locally computed gradients to said parameter server. The parameter server can then perform an averaging operation and return the gradients. After each node has applied the gradients on its local model, the training process continues and loads the next data sample. In figure 5.1a such a design is outlined using a total of $n_t = 5$ processes. One process takes the role of the parameter server while all other processes communicate exclusively via that process.

An alternative to the parameter server is an approach that is **fully decentralized**.[19, 49] Here, the processes communicate by sending their gradients directly to each other. A centralized server is no longer neccessary. The gradients themselves are calculated in the same manner as before - they are send out after the backward pass and before the optimizer is invoked. In5.1b we recreated the same scenario as before but without a parameter server. It is visible that the number of messages increased



(a) Exchanging gradients using a centralized parameter server.



(b) Exchanging gradients without a centralized server and using direct communication.

and that the number of processes was reduced. The missing process could either be saved, or also be used for training.

When comparing both solutions, there is no method that is clearly superior over the other. The parameter server can generally be considered to allow for simpler implementations while still offering decent performance. Direct communication, on the other hand, requires a more thought-out implementation as otherwise the total number of messages would scale with $O(n_t^2)$.

We will later see, that for our use-case, we only need a very small number of training processes and therefore the scaling characteristics are of lesser importance here.

5.2. Training with Online Data

Online Machine Learning *(sometimes referred to as Continual Learning or Machine Learning for Streaming Data)* is a concept where the model is no longer trained using a dataset that is stored on disk, but instead using a stream of continuously flowing data. The key difference to traditional approaches is, that past data is no longer accessible. While traditional approaches train the model multiple times using same data, this is no longer an option here. The term *epoch* is also no longer applicable, because the data is not limited in a comparable way. An implication of this is, that the model can only see the data once during the entire run and duplicates are

```
model = Model()
optimizer = Optimizer(model.parameters())
l = Loss()
group = all_processes()
while not stop_requested():
    I, GT = stream_data()                       # Step 6
    optimizer.zero_grad()                       # Step 1

    O = model(I)                                # Step 2
    loss = l(O, GT)                             # Step 3

    loss.backward()                             # Step 4
    average_gradients(model.parameters(), group)# Step 7
    optimizer.step()                            # Step 5
```

Figure 5.2.: Based on the previous code snippet, we added steps 6 and 7 which express the data streaming and parallel aspects. There is no longer a specified limited to how much data the training will consume and it will continue running as long as there is data available. The gradient exchange step (Step 7) is executed after computing the local loss and before applying the gradients. The numbering from the chapter 2 was kept.

only present due to chance.[1] As stated in [25], this makes the model prone to a phenomenon commonly referred to as *Catastrophic Forgetting*. The model, while being able to train and learn from new data, starts ‘forgetting‘ previously trained input samples.

5.3. Existing Implementations

This thesis is not the first attempt at implementing machine learning on a distributed systems, nor is it the first attempt at utilizing streaming data. In this chapter, we will present existing proposals and implementations that try to cover these topics. We will start with Melissa-DL[49, 23], a project that was already successfully used for training on a HPC cluster. Then we will cover River, a Python library that specializes on learning via streaming data. Lastly, we will look at Horovod, an established solution for distributed learning by the Linux Foundation.

5.3.1. Melissa-DL

Melissa-DL is the *state-of-the-art* implementation of distributed online learning for for high performance aplications.[49] In the past, it has been used to successfully train models on simulation data acquired from 2D head differential equations.[23]

[1]Which is effectively 0 in almost all interesting cases.

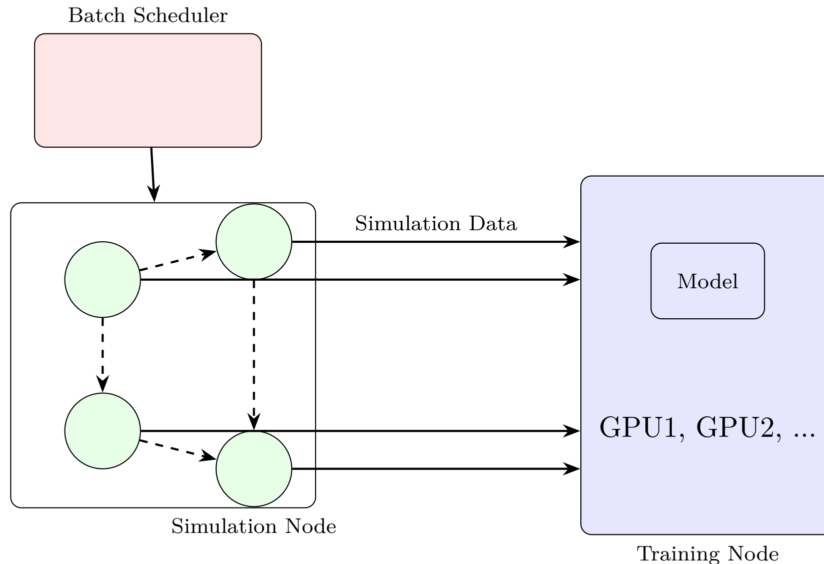


Figure 5.3.: A simplified overview of the Melissa-DL architecture.[23] The simulation processes are started by the *Batch Scheduler* and transmit their data to the *Training Node*. Missing in the picture is the *Launcher*, that is responsible for taking requests from the training process.[49][79]

A short summary of Melissas design is given in 5.3 [49]. The simulation simulation is started by the *Batch Scheduler*. The number of processes that is started or restarted is done by an external process called the *Launcher*. It takes requests from the *Training Node* and reacts upon them by requesting the according processes from the scheduler. Failed processes are restarted if needed.[49] The simulation data is forwarded from the simulation nodes via a *client-server* model to the training node, that hosts all the GPUs. This node performs all training on its GPUs and exchanges gradients using the *All-reduce* method.

Because Melissa is designed to handle fast simulations and long lasting simluation processes, it stores training samples within the training node for future access. Due to its online learning nature, it is neccessary to regularily evict data samples from this buffer. Melissa was initially tested with a *first in, first out (FIFO)* approach, but this approach showed low throughput performane during training.[49] To combat this, Meyer et al. implemented an alternative buffer eviction strategy called *Reservoir*. This algorithm achieved higher throughput during training and a validation loss close to fully offline training.[49] The reservoir algorithm assigns a flag to each sample it holds within its buffer. The flag indicates if the samples has already been used for training or not. When a new data sample arrives, the algorithm evicts one randomly selected sample from the set of already trained samples. This ensures that every sample is at least used once for training.[49] If the simulation runs significantly slower than the training node, the reservoir algorithm can reuse samples to create new, diverse batches to perform additional training steps while waiting for new

data.[49]

5.3.2. River

`River` [50] is a Python library for online machine learning models. Its origins trace back to `Creme` [50] and `scikit-multiflow` [51]. Both libraries originally started as independent and community-driven projects, but were merged into one project, which is now know as `River`[50]. River heavily depends on the Python ecosystem and focusses on simplicity and user experience.[50] For real online training, it also expects the user to handle data transfer manually via Python instead of providing an extensive framework. According to [50], one of its goal is to be used within web applications that make use of JSON payloads.

5.3.3. Horovod

Horovod is a library used for scaling existing training scripts to utilize multiple GPUs.[69] Its compatible with Tensorflow, Keras and PyTorch and has shown great improvements in regard to training performance. It was able to train Inception V3 [71] and ResNet-101 [29] with 90% and VGG-16 [70, 11] with 68% scaling efficiency. Internally it uses communication libraries like MPI, NCCL or oneCCL [48, 52, 31, 33] to perform *All-reduce* and *All-gather* operations for gradient exchange.[48] Horovod itself is a Python library as it is intended to be used in existing script, which are typically written in Python.

Both, Melissa-DL and River are projects that have already demonstrated to be capable of training machine learning models on online data. In the case of Melissa-DL this was also data that was generated using a distributed simulation software. When comparing them to the architecture already imposed by CIAO, it becomes apparent that all three are not easily integrated. Regarding Melissa-DL, the assumption is that the simulation runs fully detached from the training process and the data gets send to a dedicated node, using a client-server[72] architecture, that eventually performs the training steps on multiple GPUs. It is the goal of this thesis to enable training across multiple nodes in the same way AIxeleratorService already allows for inference on multiple nodes. Hence, the architectual design of Melissa-DL does not satisfy the requirements. River, on the other hand, does not implement any capabilities regarding distributed training which also makes it not suited for our purpose. While Horovod is able to perform these distributed training steps reliably, it is not written for online training and does not offer any features in that regard. Furthermore, all libraries are developed almost exclusively using Python while our simulation software is written in Fortran. Although interfacing these two languages is possible using FFI [7], this would induce an additional runtime cost that does not occur when implementing distributed training using online data in AIxeleratorService directly.

This concludes our chapter on training techniques for multi GPU environments and streaming online data. We started with a continuation of the previous chapter

about model training and extended it by ways that allow for training on multiple devices as well as training via data streams. We then followed up by reviewing some existing projects that try to solve at least one of these problem and explained their general approach. The concept for distributed learning has been around for some time[18, 69, 49, 23] and most of its implementation follow the same structure as we have laid out previously. They differences lie in the decisions taken outside of the core implementation and affect if an implementation is suited for a given challenge - or not.

6. Training using AIxeleratorService

Our goal is to implement and evaluate training infrastructure in the previously introduced architecture 4 This requires change within both, the solver and librares. In this chapter we will document our implementation and how it adds to the existing software. A detailed analysis will be conducted in chapter 7.

6.1. Back to MNIST

First, we want to start by laying out all changes that are required to enable distributed and online training of a model on the MNIST dataset. The model was already introduced in a previous chapter. 2 [40, 1]

For MNIST, we only need to implement changes into AIxeleratorService as CIAO and ML-Interface are only used for simulation.

To establish communication across GPU processes, as described in chapter 5, we need to build a communicator that encompassed all those processes. Within `createWorkgroups(void)` in `RoundRobinDistribution` AIxeleratorService already creates an implicit list of all GPU processes since it needs to assign non-GPU processes to them. This temporary collection can be stored permanently within a seperate MPI communicator and exported via the abstract parent class. After the creation of all workgroups, a process then has access to two communicators - one, that will allow it to pass the training data around, and a second one, that is used for exchanging gradients. The latter is only used by GPU processes.

For data exchange during runtime, we have used the `CollectiveCommunication` implementation so far. In contrast to inference, we now need to transmit more data in each step, i.e. the input data to our model and the expected output. The code for input data already exists and can be reused for the output data as well. Because we want to keep the ability to perform only inference, we need to split the functionality into two methods, such that input and output data can be communicated independently.

The new communicator layout is given in6.1. In this example, there are four training processes and a total of eight simulation processes. Each training process has a set of simulation processes associated with it, that is tasked with generating data and sending it to said training process.[54, 21] Once all training processes have received their data and started training, they can communicate their gradients via their own gradient communicator (blue in 6.1).

The inference strategy is not used for training and we will not make changes to it. Instead, we propose the addition of a new `TrainingStrategy` that operates

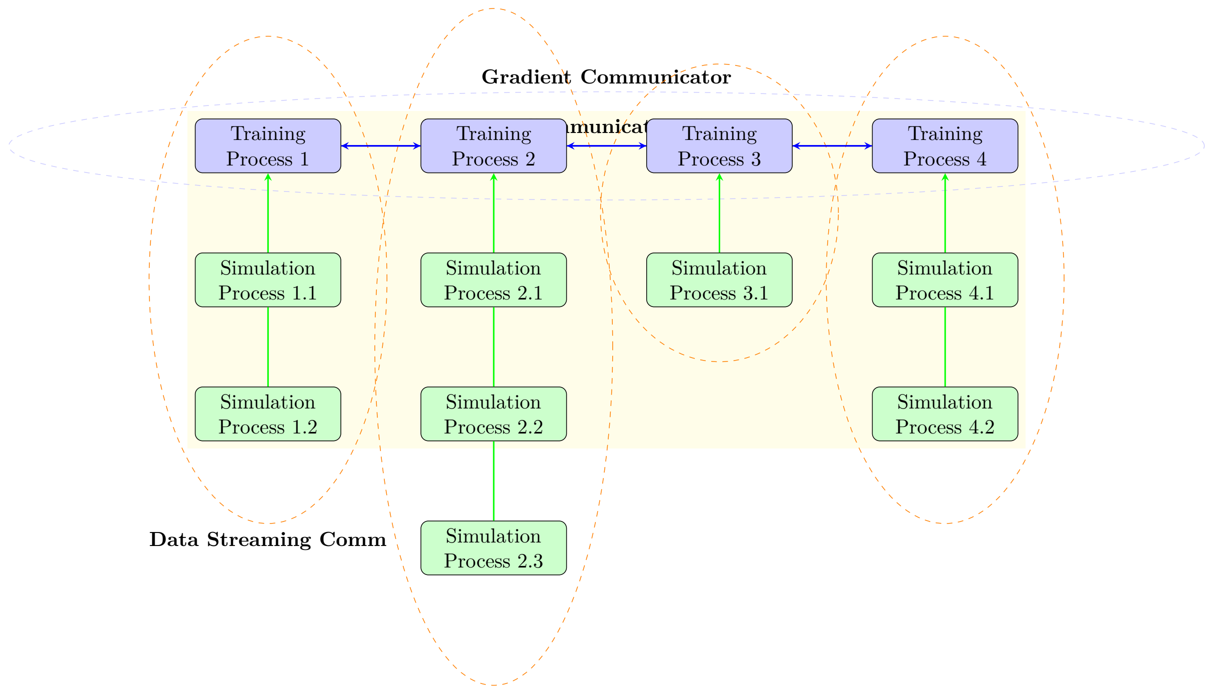


Figure 6.1.: The communicator design as it is implemented for the `TrainingStrategy`. The blue circle represents the communicator containing all GPU processes and the orange communicators are for data streaming to those GPU processes.[80]

independently. It will function similarly to `InferenceStrategy` and can be implemented for various backends, e.g. as `TensorflowTraining` or `TorchTraining`. We focus here on a Torch based workflow but emphasize that other machine learning libraries can be implemented as well.

Finally, we present the complete AIxeleratorService architecture in 6.5 with our new training strategy.

6.2. Combustion Models

Expanding on the previous section about MNIST, it is now time to also document the changes for model training on simulation data. All changes made for MNIST are also applicable here - however, instead of just extending the AIxeleratorService library, we now also have to integrate changes into CIAO and ML-Interface.

For ML-Interface we need to adapt both interfaces `ml_coupling_t` and `ml_coupling_strategy_t`. Recall chapter 4.3, `ml_coupling_t` is the interface for selecting the kind of simulation and `ml_coupling_strategy_t` is the interface for selecting the correct backend library, which is in our case AIxeleratorService. If we observe the dataflow for inference within figure 4.2 again, we observe a path starting at the specialization for combustion, that traverses via both interfaces, and

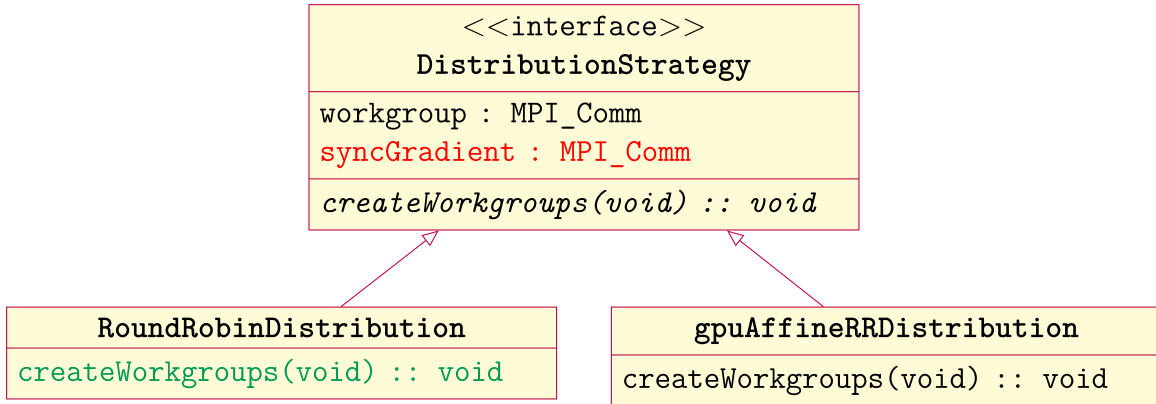


Figure 6.2.: We extended the distribution strategy [54] 4.3 by another entry, the communicator for syncing gradients. The new member is colored red and the method we needed to modify green. `gpuAffineRRDistribution` [30] was not modified and does not create an additional communicator.

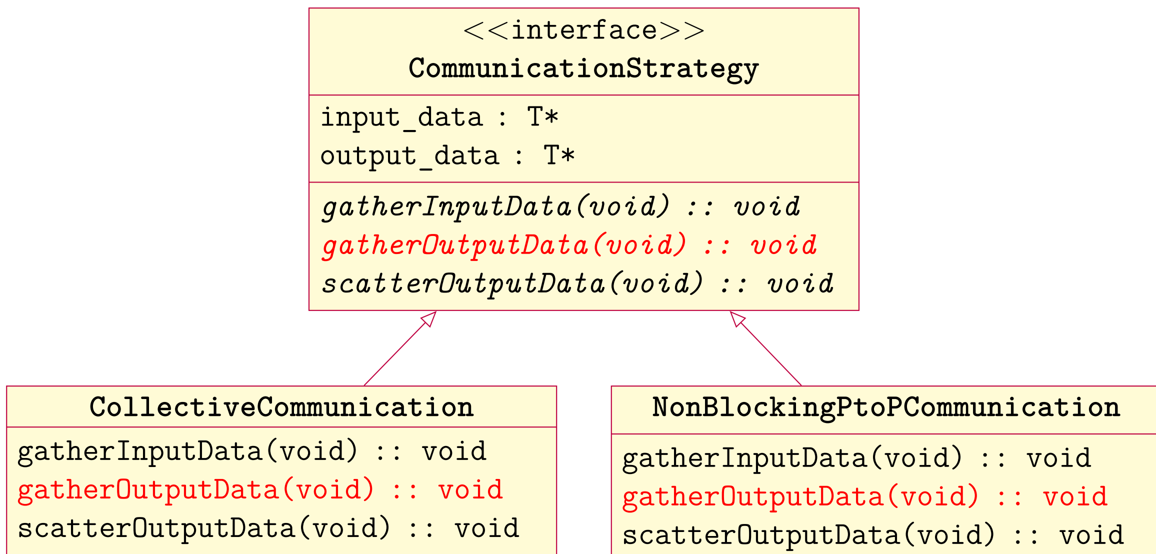


Figure 6.3.: All communication strategies are now capable of copying the output samples in addition the input data. Additions are colored in red. 6.3

finally ends at the specialization for AIxeleratorService. We need to model the same behaviour for each training step, such that we can call a specialized training method within the `ml_coupling_combustion_t` module and that leads us via the same path to the same endpoint for AIxeleratorService. Our training data, needs to undergo the same reshaping and normalization processes as for the inference case, with the addition that we need to process two arrays now instead of just one. At the same time, we do not need to transform the output data back to be used in CIAO like it is done for inference. To summarize, within ML-Interface we need to replicate the

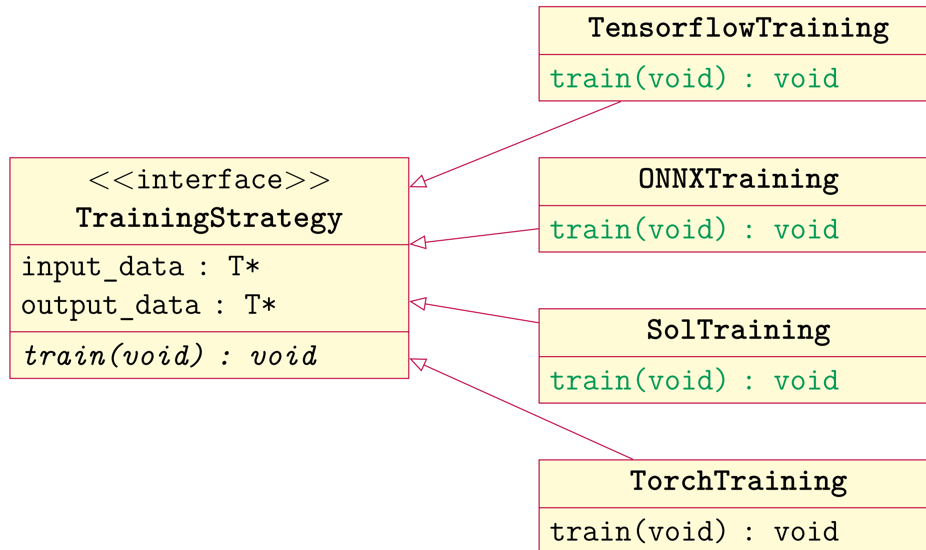


Figure 6.4.: The newly proposed training strategy for AIxeleratorService. It provides an abstraction for selecting between multiple libraries such as Tensorflow, ONNX and Torch. In green we highlighted methods that we suggest but did not implement.

design used for inference while changing the direction the output data flows.

In figure 6.6 we present the new changes to the interfaces of ML-Interface together with their implementation. The training data takes the same path as it does during inference.

CIAO needs the least amount of change. We initialize the ML-Interface library which will then initialize the AIxeleratorService. Afterwards, CIAO only calls into ML-Interface after each simulation step. It will write the current state of the simulation into a preallocated buffer that is accessible from ML-Interface and then order ML-Interface to perform the training step using AIxeleratorService.

6.3. The TrainingStrategy

The new `TrainingStrategy` is initializes similarly to the existing `InferenceStrategy`. While the `InferenceStrategy` parses its input and produces an output, the `TrainingStrategy` consumes an input and an output. In contrast to the `InferenceStrategy`, it produces no output. The raw data is paired with the intended dimensionality and converted into PyTorch tensors. Since the data can exceed the size `64x64`, it is trimmed to the appropriate size. After the forward pass through the model is performed, the loss is calculated using a predefined criterion. This result is then used for the backward pass through the *computational graph* to using `loss.backward()` to calculate the gradient for each operation. If we are training on the GPU, it is now required to move all data and the model back to main memory. This is done by targeting the `torch::kCPU` device. For purely CPU based runs, this step is not

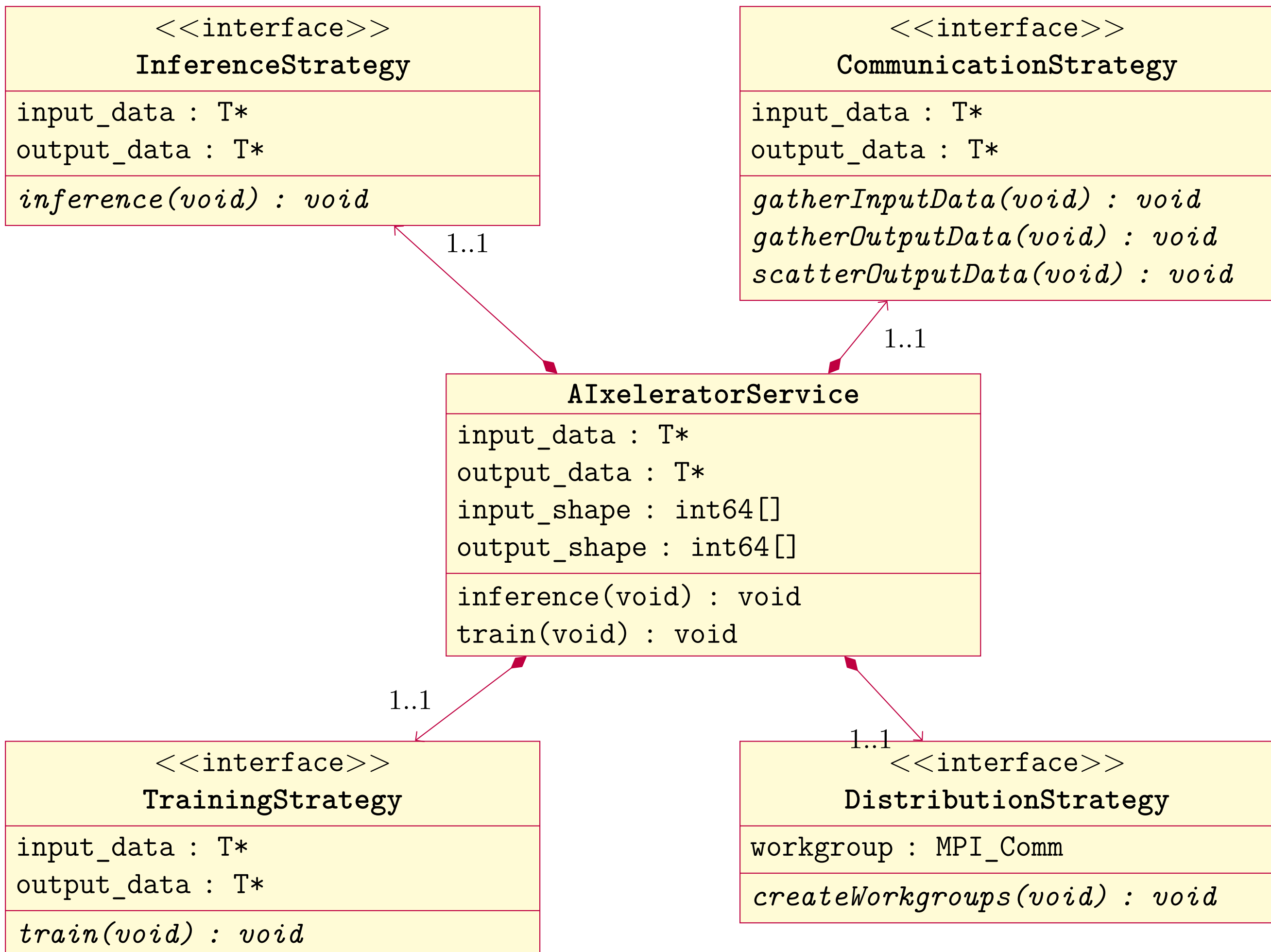


Figure 6.5.: A complete overview of the AIxeleratorService library [54] with all its components.

required. This enables to use the memory address from the internal gradient tensor directly from `MPI_Allreduce`[48] and perform the summation step *in-place*. The communicator for this is the new communicator from 6.1. Once all gradients are exchanged, the training is resumed using the optimizer to copy the new weights to the model.

To prevent rare deadlocks, it is important that all training processes exchange the total amount of iterations they are able to train. The limiting factor here is the amount of data received from the simulation. Because the amount of simulation processes assigned per training process is not constant, it can happen that some processes already finish while other attempt to train one more round. Since we include all training processes within one MPI communicator, this would result in a complete freeze of the affected training processes. In addition to that, CIAO waits for all processes to complete for starting the next iteration of the simulation, which results is no further progress on any process. To combat this, it is important to

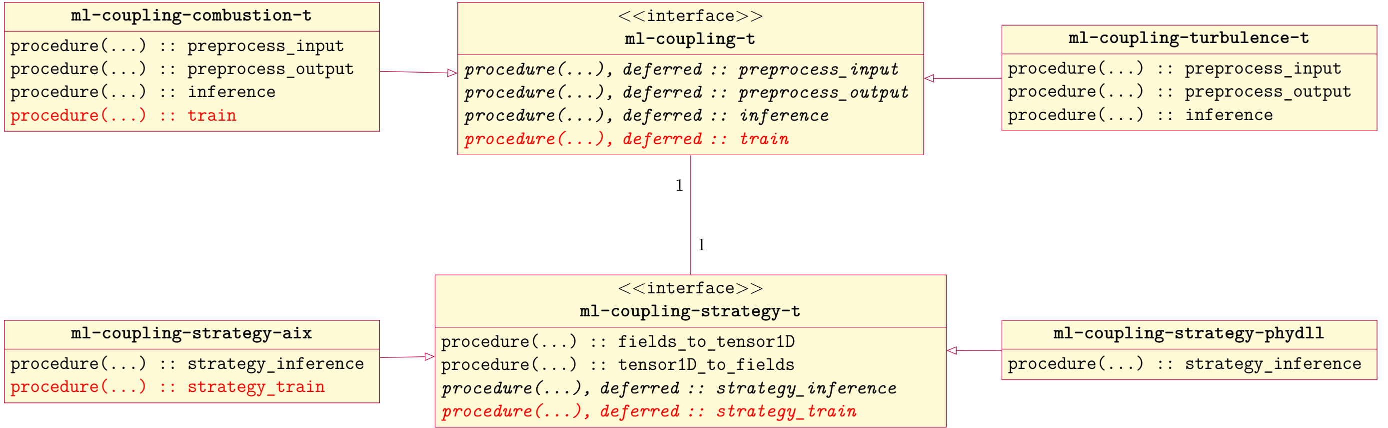


Figure 6.6.: The architecture of the ML-Interface library [54]. For clarity we have omitted types and functions used internally. *Italic* functions are abstract and implemented by the respective children.

determine the maximum number of steps the training processes should consider. This is done using a prepended, seperate `MPI_Allreduce` operation that utilizes the `MPI_MIN` operation.

6.4. Ensuring Integrity

Our overall implementation has gotten significant complexity. The data is stored in multiple places and copied around frequently. In between those copies, it is transformed and reshaped depending on factors only known at runtime. It is no longer trivial which process is communicating with which other process and this could lead to erroneous data being pushed to the GPU.

One way of avoiding this is using a hash algorithm[62, 15]. A hash algorithm will map objects, usually represented as numbers, to seemingly random numbers from a smaller domain.

$$h(\cdot) : \{0,1\}^m \to \{0,1\}^n$$

with $m > n$[62]. These hash algorithms try to avoid mapping numbers that are near in $\{0,1\}^m$ to numbers that are also close in $\{0,1\}^n$[62]. The result is a mapping that looks almost random.[1] Within AIxeleratorService, a hash algorithm ensures the integrity of data throughout the entire library. The data is hashed after it has been received from ML-Interface and then hashed again just before it is send to the GPU for training. That means we can verify the correctness of both strategies for distribution and communication. The hash algorithm is provided by the library

[1]Some hash algorithms also provide functionality to be computationally hard to reverse, but this is not needed for validating data integrity.

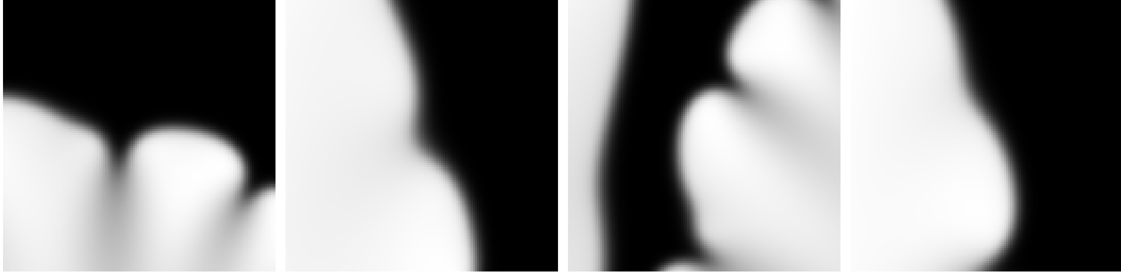


Figure 6.7.: Data extracted from AIxeleratorService using the described BMP[44] format. Bright regions correspond to large values for the progress variable and dark regions to lower values.

`xxHash`[15] and completes the SMHasher test suite[15]. Using this library, it was possible to verify the data integrity when training with MNIST data as well as data from CIAO.

Using a hash algorithm is unfortunately not possible for the entire training process. The main reason is the data normalization that occurs within ML-Interface. Scaling the data causes it to be mapped to a different value. Because all our data can be visualized, i.e. MNIST already consists of images and the simulation data can be visualized based on the progress variable as we have done in chapter 2, it is possible to extract the data at different stages and verify it visually. A suitable format for this is BMP[44]. It adds no compression, has minimal computational overhead and can be used to write almost a plain bytestream to disk.For the purposes of this thesis, AIxeleratorService has the ability implemented to create a BMP image from a tensor and save it to disk. This is an essential tool for verifying the integrity of our data across multiples libraries and transformations as images will retain their structure despite normalization. Some of the resulting images are depicted in 6.7.

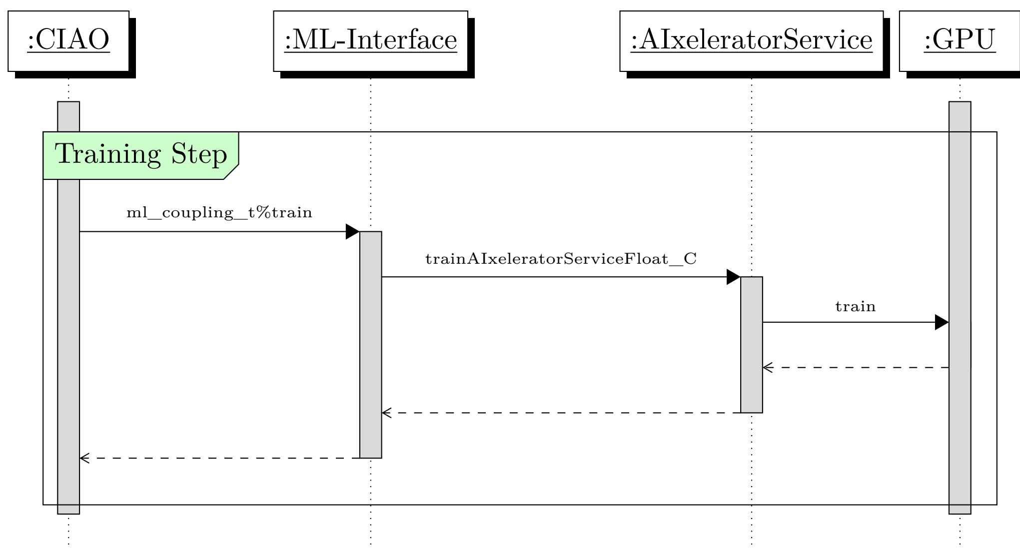


Figure 6.8.: An abstract view of the training process. Function parameters are omitted for readability.

7. Analysis & Evaluation

In the previous chapter, we gave an overview of the training implementation. We will now move on and analyse the its performance characteristics in regard to model training capabilities as well as its computational performance. We will highlight the different steps along one full iteration of the training process and provide measurements on their impact. Finally we will conclude this chapter by identifying bottlenecks within our training process. MNIST will be the leading example, followed by the UNet for simulation data.

7.1. Model Performance

We will begin our evaluation of model training performance. For MNIST, we have the following metrics to analyze:

Training loss This is the loss that is calculated after each forward pass through the network. It is based on the training data set and repeats periodically after all 60000 samples have been used. This loss is expected to reduce over time simply because the model repeatedly sees the same data. If this is not the case, the model does not learn from the data and presents as an indicator that there is a fundamental problem with either the code or the model. A problem regarding the code could happen during data copying. In this case, we copy invalid data from memory and try to train the model on it. Becaues this data has no inherent structure, the model cannot make any progress in learning from the data. The result would be a fluctorating but near constant training loss. Another problem that can lead to a problematic training loss lies withing the model itself. If the model is simply too weak, it can also not learn from the training data because it does not posses the capability to produce correct output. One example for this, are model that do not have enough parameters for the problem at hand. However, in chapter2 we have already demonstrated that the model is capable of learning from the MNIST dataset. Therefore, the only source for error here is incorrect data handling before the training step. In figure 7.4 we have depicted the training loss when training the MNIST model using AIxeleratorService on multiple GPUs and online streaming of

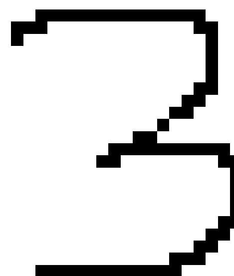


Figure 7.1.: A correctly identified digit that was not part of the training dataset.

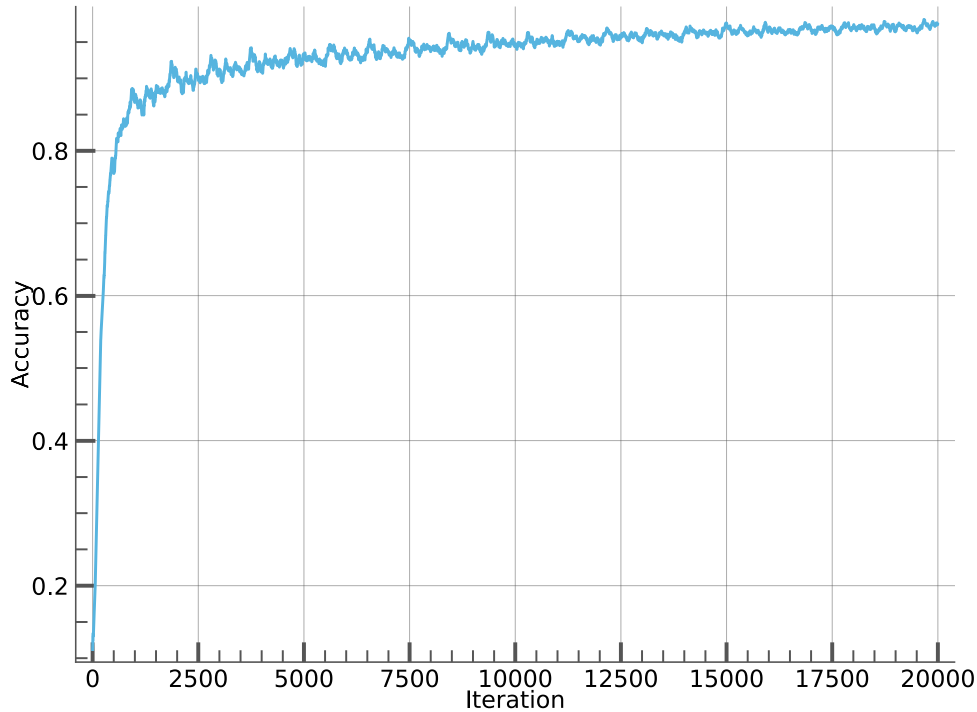


Figure 7.2.: MNIST training accuracy

the data. It is appearant that the loss is shrinking over time - especially on a logarithmic scale. We can therefore come to the conclusion that our training pipeline works correctly.

Another very important metric is **accuracy**. It measures the percentage of correctly labeled samples compared to all predictions made. Accuracy can be used in addition to the training loss to evaluate the training progress of the model. The formula is:

$$\text{accuracy} = \frac{\text{C}}{\text{A}}$$

with C being the number of correct predictions and A the number of total predictions. The range is from 0 to 1 and values closer to 1 are considered better. For our model training on MNIST, we achieve an accuracy value depicted in 7.2. The figure shows that our accuracy is increasing over time and converges towards 1. Towards the end of our training, we reach values of up to 0.976 meaning that 97.6% of samples were classified correctly.

One more possible metric we could consider here is the **test loss**, that is collected by evaluating the model on portion of the dataset that it has not seen yet. Typically the dataset is split by some fixed percentage beforehand. Since we already showed

in chapter 2 that the model for MNIST does not overfit, we will not use this loss here.

Finally we want to confront the MNIST model with a digit that was not drawn from its training dataset. Depicted in 7.1 is a hand drawn number 3. Using the saved model weights from the model after training, we can verify that the model indeed is capable of identifying the number.

We will now continue our evaluation of the training process with the UNet++ model on the simulation data generated by CIAO. We use the same techniques as for the MNIST case and, overall, we expect a similar behavior when training. We have previously established in chapter 2 that the model is capable of learning simulation data. In addition to that, when training the UNet++ on live simulation data, we do not have a test dataset and in contrast to MNIST, we cannot expect to receive the same data twice. While the MNIST dataset would start at the beginning after it has reached the end, the simulation will continue to create new data. Furthermore, all data is lost after it has been overwritten in memory - meaning that we cannot access previous data samples. Creating a seperate dataset for testing is therefore not the correct approach to this, as this is already done continuously by the simulation process we use for the training data. Our only evaluation metric for model performance therefore is training loss. The usual problem of overfitting does not apply here due to the previously stated characteristics.

Evaluating the UNet++ on simulation data, we get the results shown in 7.4. And even though we do not repeat data, unlike with MNIST, a repetitive pattern becomes visible within the training loss. Using the logarithmic representation, it is also visible that the pattern only occurs after the first 50 to 100 training samples ruling out any problem with the code that performs the training steps. The overall trend is, again, a decreasing loss with time, which is what we expect.

The cause for this pattern is the segmentation of the `2048x2048` field into smaller subfields.2 The combustion process does not happen at all locations at the same time but rather moves across the entire `2048x2048` field while simulating it. At the beginning, no combustion has taken place and there is no H_2O is present. While the simulation progresses and hydrogen reacts with oxygen, H_2O is formed. This causes a change in the progress variable 2, i.e. changes the entries of the `2048x2048` field. While is chemical change advances forward, it leaves behind a region of molecules that have already reacted and faces molecules that have not yet reacted at all. When CIAO splits this field into subfields for each process, it is inevitable to carve out subfields that lie at the very beginning or end of the simulation. These subfields will consist of mostly the same data - either a high or low progress variable. Learning to predict these fields is an easy task for a neural network as they are almost identical to their input and the distribution lies very close to some mean value. To explore the extend of how many subfields fall into this category, we analyze the variance of each subfield. Subfields that are rich in structure and contain valuable information such as in6.7 will have a higher variance. They display the current front of the chemical reaction and contain values for the progress variable that are far apart - some parts have already reacted while others have not. Measuring the variance of

the input fields before they get passed to the GPU, we can draw a picture of the distribution of our simulation data. 7.3 The graphic immediately shows that over 1000 samples, a huge majority, has a variance close to zero. These samples are hold essentially no information and all entries regarding their progress variable center closely around some mean value. Overall we can observe three clusters in regard to variance distribution: One on the left and clearly the largest. It is strictly below 0.05 and is surrounded by a large area that contains no samples. Next, we can observe a small cluster in the middle, around 0.11, that ranges from about 0.09 to 0.13. Besides it lies the last cluster and its the one that itself has the highest variance. Its range is the largest going from 0.15 to 0.24. All 2479 input samples fall into one of these clusters and it is obvious that the distribution is not even but rather right-skewed. This explains the repeating pattern in 7.4. The model is presented repeatedly with similar data that has low structural information. Training on this type of data is significantly easier that on data that is richer in information and lies further to the right in 7.3. Performing training steps over and over on this type of data does not improve the quality of the model but instead wastes time and computing resources. Even further, it could train the model in such a way that it develops a bias towards low-information data. Such bias would then decrease the quality of samples generated for other subfields as well. To prevent training on subfields that show no sign of any chemical reaction, we apply a filter, that can skip certain samples during the training process. Ideally we want to even out the clusters identified before and train our UNet++ on each cluster with an equal amount of samples. One approach to this is to store the a set of samples, sort them into clusters and then train our model on each cluster in a *round-robin* fashion. An easier approach to this is to use chance when deciding whether or not a samples should be used for training. The probably for each samples to be used for training should be non-zero as do not want to exclude any group of samples from the training process. This would again change our data distribution to such a degree that the model quality would suffer. Motivated by the law of large numbers, randomness could instead be used to create a heuristic that approximates the cluster based filtering. From 2479 total samples, we have enough information to infere the likelihood that a given sample is part of a certain cluster. For the largest cluster, that contains the most number of samples, we obtain a prior $p(\mathrm{C}_1) = \frac{\#\mathrm{C}_1}{2479} = 0.76079$, meaning that from all samples 76.079% fall into this cluster. For the other two clusters we obtain $p(\mathrm{C}_2) = \frac{\#\mathrm{C}_2}{2479} = 0.1085$ and $p(\mathrm{C}_3) = \frac{\#\mathrm{C}_3}{2479} = 0.1189$.[1] From those priors, we can now create a filtering algorithm, based on rejection sampling [74], that smoothes the distribution of our input data. For this, we generate a random floating point number r between 0 and 1. If our sample falls into cluster i and $r > p_i$ holds, we reject the sample. p_i is defined as

$$p_i = \max_k \frac{p(\mathrm{C}_i)}{p(\mathrm{C}_k)}$$

[1] This does not add up to 1 because the data contains some outliers that were chosen to be ignored here.

The `max` operation ensures that we always pick the cluster with the smallest number of elements for the denominator. Otherwise the algorithm would also start rejecting samples from the smallest cluster. AIxeleratorService evaluates the information content of a given sample and then calculates a variance metric based on the input tensor. This value is then compared with regard to the previous rejection condition and depending on that AIxeleratorService continues with the training process or skips to the next sample. Overall it is more likely to skip samples that fall into the cluster with low variance but all clusters will be considered during training. In figure 7.8 we present the training loss that our UNet++ model achieves. We can observe two things: *1)* The repeating pattern from 7.4 is much no longer present. There is some pattern between samples 150 and 400 but it vanishes as the model continues training. *2)* The model achieves a slightly better training loss than without sampling the input data. This can be seen when comparing it with figure 7.4. There the model can reach lower loss values but when considering the peaks in that graph, we observe an improvement. Those peaks represent the data with higher variance as it is harder for the model to converge on those inputs. During the entire training run, we rejected a total of 859 samples and trained on 1625 samples. This makes a rejection rate of 52.86%. Of course, these samples had to be generated, and therefore this approach roughly doubled our runtime.

Comparing the our implementation to the baseline, that is the offline training from chapter 2, we have to come to the conclusion that our model performs worse on fresh data. While the offline model, achieved a loss of 4.7×10^{-4} within the first 2479 data samples, our online model does not go below 1×10^{-2} on extended runs, making our model worse by a factor 10. This leads to a direct comparison with Melissa[49], where this behaviour did not occur when using the *Reservoir* algorithm. They were able to reproduce offline trained models using online data. Our conclusion therefore is, that our model experience the *catastrophic forgetting* effect described by Tao Feng [25].

7.2. Computational Performance

We will now follow up with a discussion on training performance. Regarding MNIST, we trained the MNIST model on the CPU at first, and then on the GPU. The results for GPU training are presented in 7.5 and those for CPU training in A.1. The time spend on the forward and backward process is greatly reduced for GPU training. While both were previously the dominant factor contributing about 89.1% to the overall time spend on training, GPU training reduces this to 8.5%.7.6 The dominant factor for GPU training is the communication overhead. It comprises exclusively of time spend on gradient synchronization. As we have discussed before, after each backward pass and before each step of the optimizer, we average the gradient across all processes. Comparing figure 7.5 with A.1, we can even observe, that the absolute time spend on communication increases when performing training on the GPU. We achieve a runtime of $(7.031\,000 \pm 0.004\,337)$ ms for communication when training

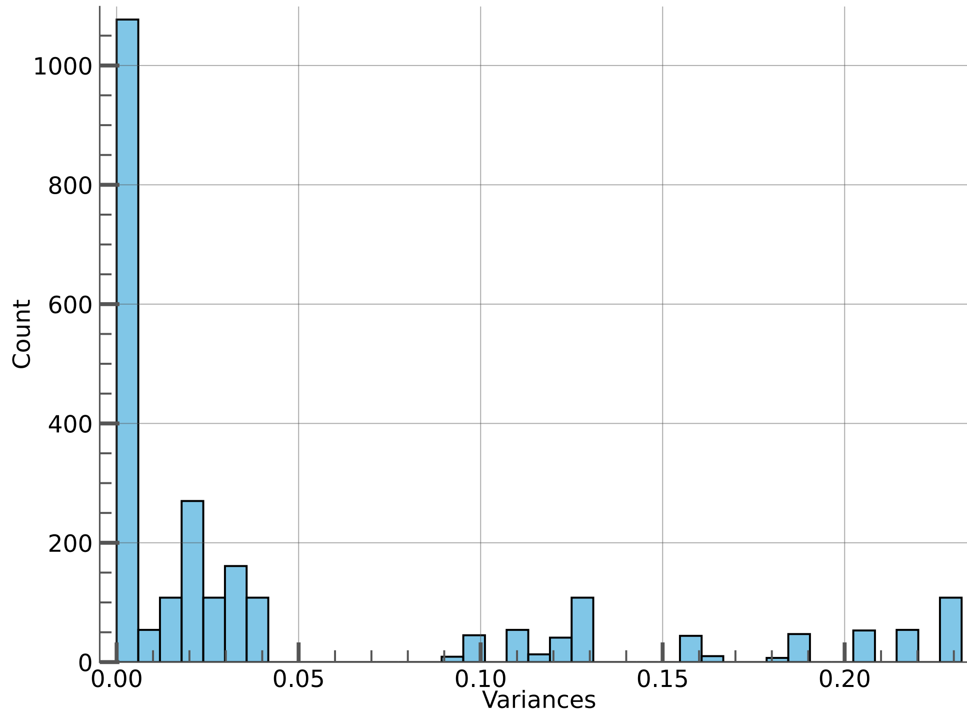
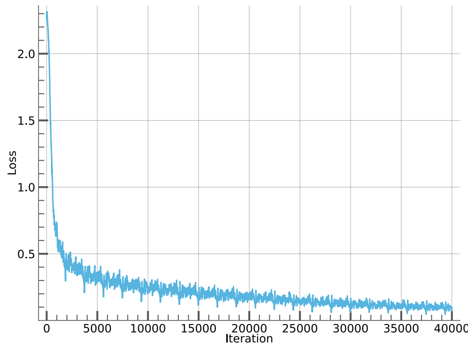


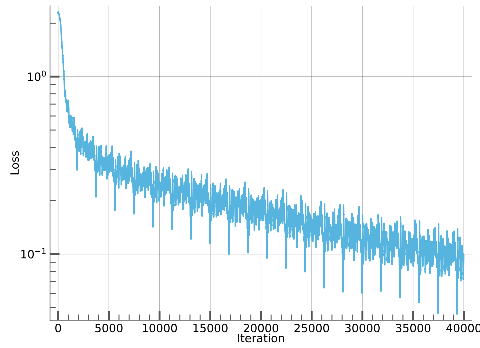
Figure 7.3.: A histogram showing the distribution of the variance of 2479 samples.

on the GPU and $(6.636\,00 \pm 0.014\,53)$ ms when training on the CPU. It is clear that this difference cannot be caused by measurement uncertainty. This increase in runtime is caused by additional memory copies to and from VRAM that are not present in the CPU case. When synchronizing gradients, we utilize `MPI_Allreduce` to transmit the gradients across all nodes within the communicator. Although we already perform the operation *in-place*, the memory location used by MPI here has to reside in main memory. When training on the CPU, the gradients can be directly written into the buffer allocated for the tensor, but with GPU training this is not possible. We first need to copy the tensors from VRAM and only then we can perform the `MPI_Allreduce` operation. Afterwards, it is also neccessary to copy all tensors back to VRAM again, so we can continue with our training step.

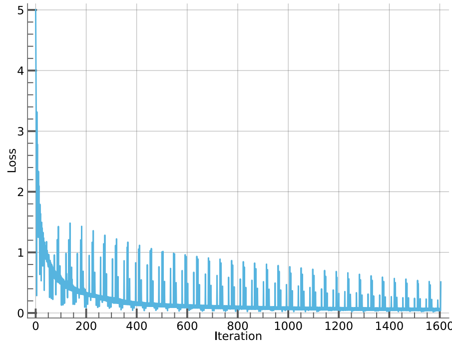
Overall, it is not surprising that GPU accelerated training is faster than training a model on the CPU and the increase regarding communication is only minor compared to the performance gains for the forward and backward pass. The superiority of GPU training compared to CPU training has been demonstrated countless times.[37] What matters to us is that we present an implementation that is correct and efficient.



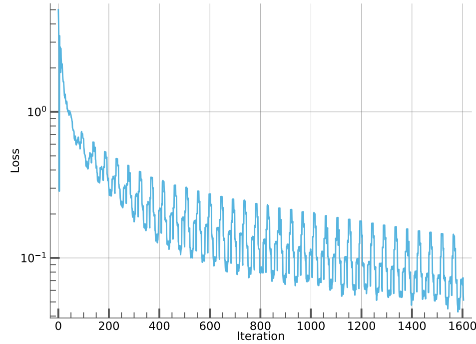
(a) MNIST training loss



(b) MNIST training loss (logarithmic)



(c) UNet++ training loss

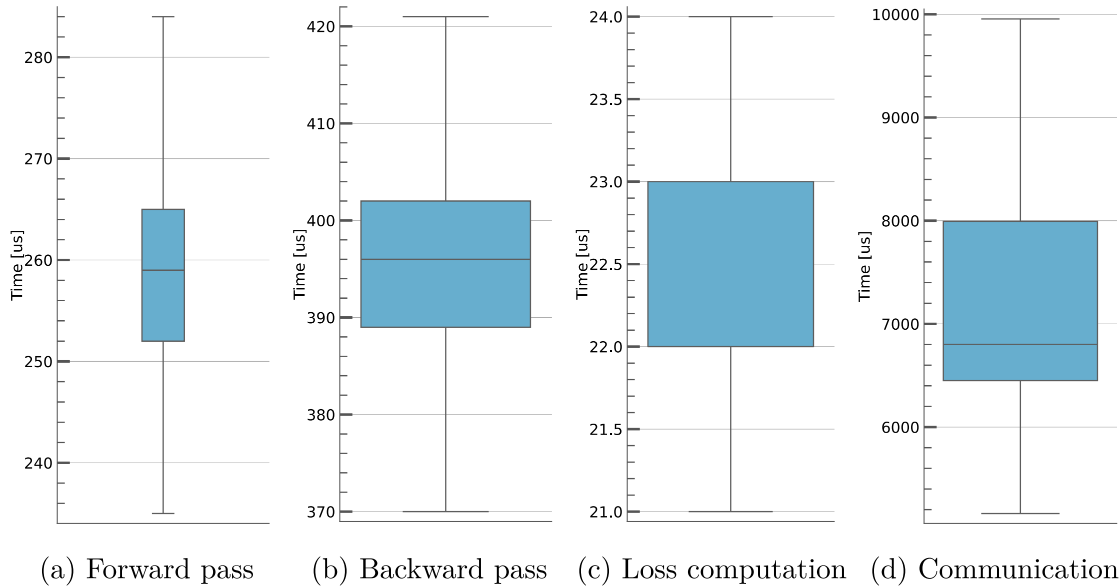


(d) UNet++ training loss (logarithmic)

Figure 7.4.

Training the UNet++ using simulation data from CIAO, the initial forward process took (133.598 ± 0.091) ms followed by a computation of the loss with (3.772 ± 0.002) ms and a backward pass (258.099 ± 0.039) ms. Before applying the gradients, they need to be exchanged between all participating GPU processes. This communication step introduces some overhead, which we measured to be (30.885 ± 0.208) ms. Comparing all those values, we get a picture like 7.6. It is apparent, that the backward pass dominates the overall runtime of our training process - directly followed by the forward pass. When we move our model and all its data on the GPU, we get the results presented in 7.5. The performance improvement is obvious, by using the GPU for training the overall share taken by the forward and backward process is significantly reduced. Before it consumed about 92% of each training step, GPU training reduced this to 18.6%. A result that is similar to what we observed in the MNIST case and expected as GPU training has been done many times for this purpose.[37]

Unlike our MNIST model, which relies on an *in-memory* buffer for data loading, UNet++ needs to wait for CIAO to complete one simulation step. With our sam-



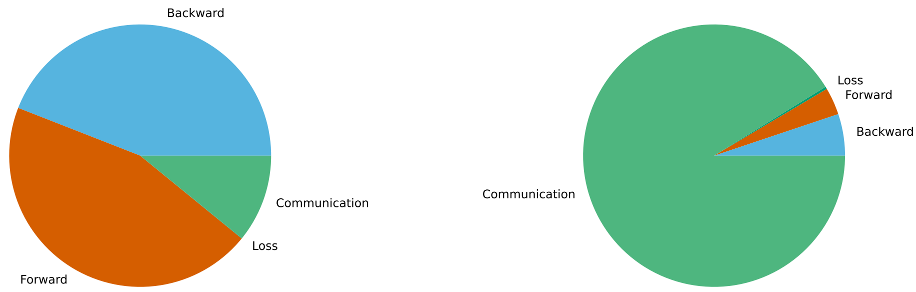
(a) Forward pass (b) Backward pass (c) Loss computation (d) Communication

Figure 7.5.: Benchmarks when training our MNIST model on the GPU.

pling algorithm from before, this time is expected to be doubled. During our measurements, CIAO spend $(97.639\,400\,0 \pm 0.077\,022\,6)\,\mathrm{s}$ simulating each step.[2] This makes CIAO significantly more time consuming than any other procedure in our training process. CIAO takes 37×10^3 times longer than the forward pass on the GPU and730 times longer than that same forward pass executes on the CPU. This marks the simulation as a clear bottleneck when training a model such that we spend 89.06% of our total runtime simulating fields in CIAO. On top of that, this calculation does not account for rejected samples, which would increase our total simulation time even further. The pie charts in figure 7.6 for UNet++ do not include time spend within CIAO. For this purpose, we present **??**. We have to come to the conclusion that GPU-accelerated training does not play a role as significant as it does when training a model on MNIST. Outsourcing the actual training onto a different thread an resuming the simulation immediately, would eliminate all benifits of training UNet++ on a GPU.

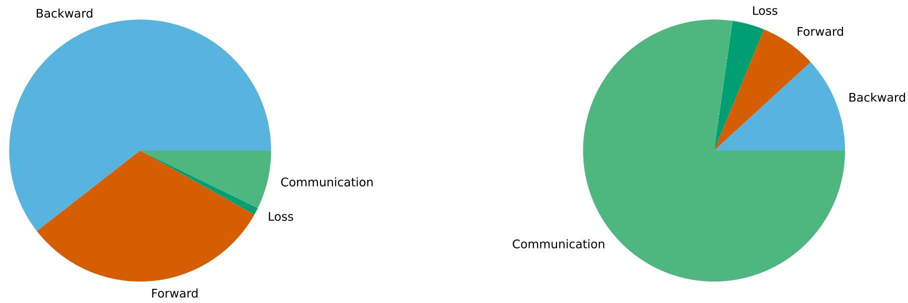
In this chapter we have proven that our implementation is capable of training a model on the MNIST dataset and a UNet++ model on simulation data. Both training processes were done on multiple GPUs in a distributed fashion along with online data streaming. It was not required to save any data to disk during the entire training procedure as we have done in the introduction. For MNIST, this property of not saving data is not important as the entire dataset is only a few MBytes large and fits into main memory of almost all modern computers. It would even be possible to store the entire MNIST dataset directly in VRAM. MNIST served here as a proof of concept that training using AIxeleratorService is possible. We have a good understanding of MNIST and can visualize the dataset easily, but when

[2]Note the change in units here.



(a) A comparison of all steps involved when training MNIST on the CPU.

(b) A comparison of all steps involved when training MNIST on the GPU.

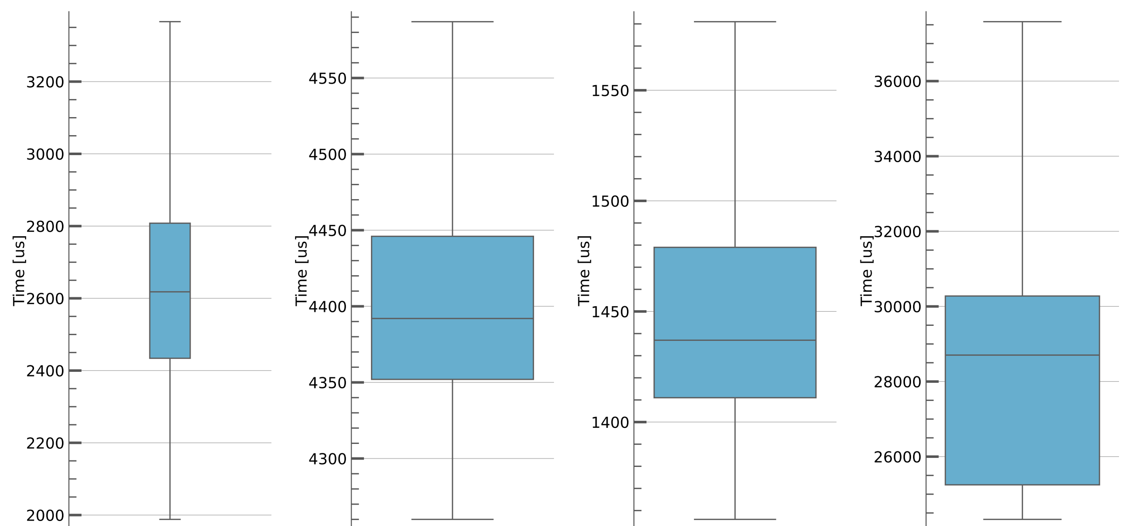


(c) A comparison of all steps used for training UNet++ on the CPU excluding CIAO.

(d) The same steps as in the CPU executed on GPUs, again without CIAO.

Figure 7.6.

we transition to simulation data for the UNet++, this is no longer the case. The training data is no longer as trivial and we have no reliable reference point that allows us to judge wether or not our implementation is working. Our successful attempts at training the MNIST model allowed us to progress further and switch the MNIST dataset with CIAO and the model with the UNet++.



(a) Forward pass (b) Backward pass (c) Loss computation (d) Communication

Figure 7.7.: Benchmarks when training the same UNet++ model on GPUs.

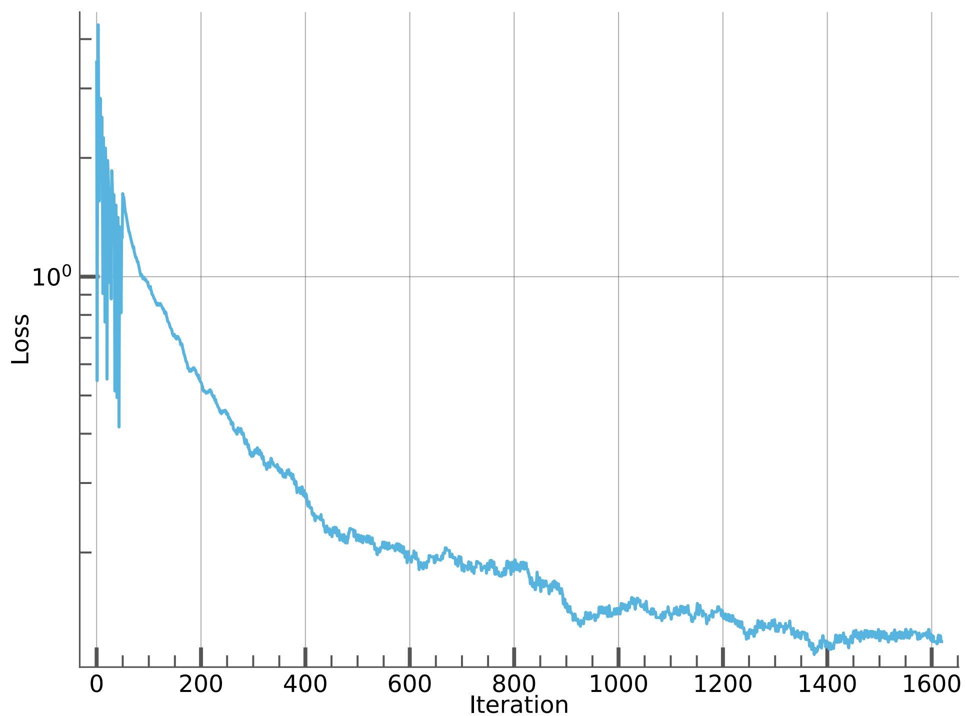


Figure 7.8.: The training loss of our UNet++ on simulation data after applying variance based filtering.

8. Conclusion

This leads us to the end of this thesis and, looking back, we will now give a brief summary of what we have done and the results we obtained. We started this thesis with the initial problem that disk bandwith of modern computer systems is not designed to transfer the amount of data needed for training models that rely on large volumes of data. After a short benchmark of the CLAIX2023 system, we came to the conclusion that the step of saving data to disk should be avoided. This itself was not a new idea, as we have stated.5[49, 23] But for our presented use-case, a UNet++ model trained on simulated hydrogen combustion data, this has not been done before.

We followed up with an introduction to the model we want to train and the structure of our simulation data. Additionally we established another example along UNet++, which was the MNIST database and a corresponding model. Because MNIST is simpler but still similar to our original example, we presented it to provide a *proof-of-concept* of our implementation.

Afterwards, we reiterated the basics of model training with a special focus on how gradients influence model updates. For this, we used the backpropagation algorithm as well as a simple variant of gradient descent. Combined, both algorithms are capable of performing model weight updates based on input and expected output data. We concluded this chapter with an outline of how a basic training algorithms looks like. This algorithm did not yet feature the characteristics we mentioned in the introduction.

Before we then continued extending this to distributed and online learning, we took a tour of the applications and libraries we would be using for implementing the training steps. We introduced CIAO, ML-Interface & AIxeleratorService and detailed how they relate to each other for simpler inference applications.

Eventually, we returned to the concepts of model training and extended the previously established foundations by covering distributed and online learning approaches. It was outlined how they differ from classic training pipelines and how we need to setup communication within our process to enable distributed training.

Following this, we extended ML-Interface and AIxeleratorService to allow for training with data from CIAO. A new strategy was added to AIxeleratorService that is soley responsible for training. We highlighted to the entire training process works and how the data is passed from CIAO to AIxeleratorService via ML-Interface. To ensure correctness of our implementation, we employed a hashing algorithm within AIxeleratorService to validate that we our data handling is without any mistakes. For parts that cannot be verified via hashing, a image library was used to generate visualization from the training data. This was possible since the transformations

we applied to the data kept the internal structure such that visualizations like this were possible.

We then concluded with an analysis and evaluation of our implementation regarding model and computational performance. At first, we showed that both models are able to train within our program and, given enough data, will eventuall converge. We analyzed the training loss and, for our MNIST model, also the accuracy. For our UNet++, there was no need for an additional test dataset since all samples were already newly generated. Whilst evaluating UNet++, we made the observation that the model train mostly on data, that holds little information. This was identified by calculating the variance of each sample while training. We used a filtering algorithm to sort out samples, that bring too little value to the training process, with some probability. This caused our training loss to stabilize and we were even able to achieve a slightly better result using this method. On the contrary, this approach doubled the amount of simulation steps since we disregarded over 50% of all samples. In the second part of that chapter, we conducted a runtime performance analysis of our implementation. Our results showed that while GPU training overall greatly improves training performance, the communication needed for gradient exchange increases. For models with quick data access, like with MNIST, this increased communication cost was negligible and we gained a considerable speedup. On the other hand, when training UNet++ on simulation data, we observed that the overall runtime was completely used by CIAO to simulate data samples. CIAO consumed 89.05% of the overall runtime when training on the CPU and 99.99% when training on the GPU. This showed that GPU training, when combined with CIAO, does no longer beat CPU training as soon as training is outsourced onto an external thread.

Unfortunately, our model was affected by the *catastrophic forgetting* symptom that often arises during online training. The resulting model quality was therefore decreased compared to an offline model.

We were able to present a working implementation for our use-case and demonstrated its correctness and efficiency by training two models with it. Our implementation trains the model on data it receives during runtime and does not need to read it from disk. Thereby avoiding the I/O bottleneck mentioned at the beginning. However, our implementation is limited by the performance of the simulation used and regularly needs to wait for new data. The training processes communicate to enable distributed training, such that each of them handles their own data but the same gradients get applied to all model instantiations.

8.1. Outlook

Building on this thesis, there are many ways to continue. We have seen that simulation performance is holding back the overall training process. Further research in this field could lead to more data for model training. The simulation process can not be accelerated arbirarily by using more nodes each time due to the overall limit

in pixel a field can posses. However, one solution to this could be to start multiple independent simulation to generate data. This way, the simulation overhead could be further reduced and only limited by the amount of cores available. Furthermore, data augmentation [58] steps could be taken to artifically increasing the amount of available data without risking overfitting. Due to the nature of combustion simulations, it is plausible to assume that a mirrored output is just as valid as the original and therby doubling the total amount of data.

To combat *catastrophic forgetting*, it appears neccessary to fully implement buffers that mark and store previous elements the same way Melissa did. This provides more data to combat the slowdown caused by CIAO and would at the same time increase model quality.

Our implementation used MPI[48] to exchange gradients. For GPU-based tasks this meant we needed to copy all data from the GPU back to main memory. MPI was chosen because it was the technology already used in CIAO and AIxeleratorService. Future work could improve on this and implement gradient exchange via CUDA aware communication interfaces such as NCCL. [52]

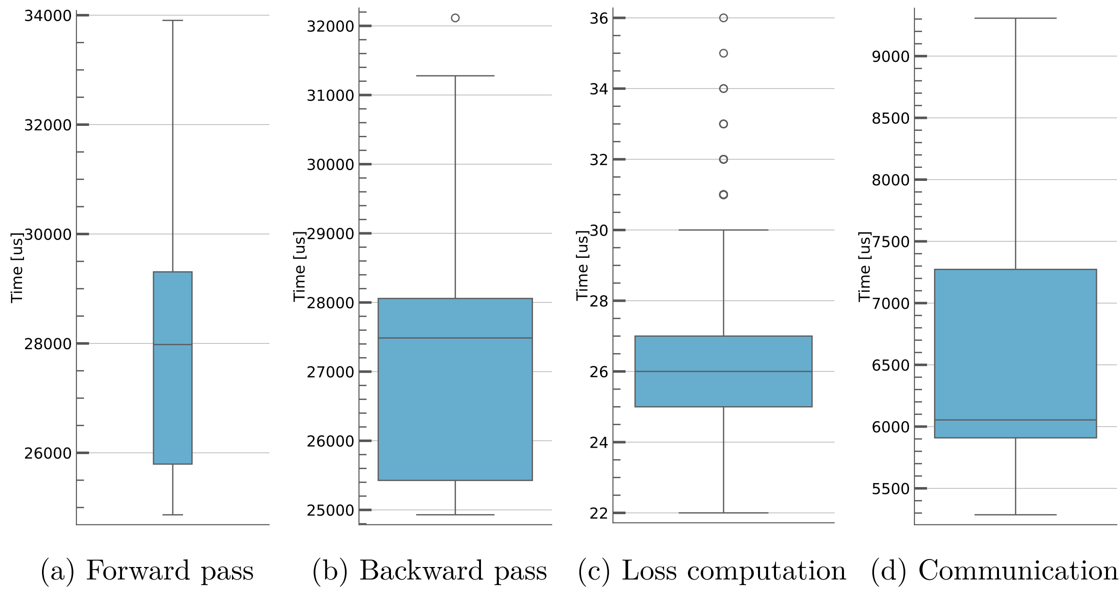
A. Appendix

A.1. Hardware used for Benchmarks

We used the CLAIX2023[67] cluster for training.

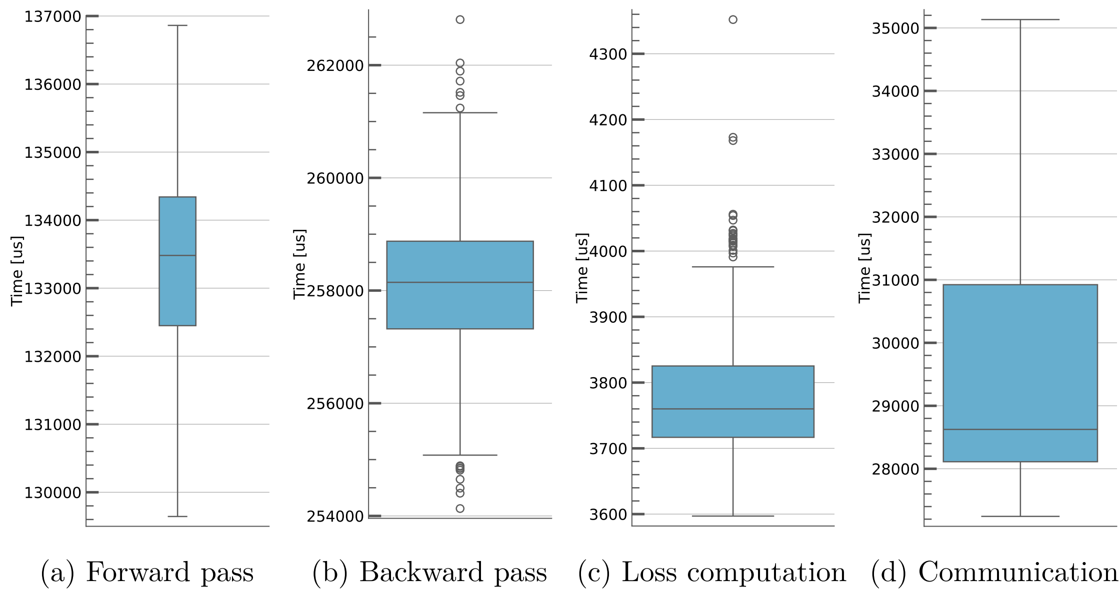
Attribute	Description
GPUs	4x NVIDIA H100 96 GB HBM2e
CPU	2x Intel Xeon 8468 Sapphire (2.1 GHz, 48 cores each)
Memory	512 GB
Interconnect	NVLink, InfiniBand
Storage	635 GB local SSD

Table A.1.: Specs for a single node with one GPU



(a) Forward pass (b) Backward pass (c) Loss computation (d) Communication

Figure A.1.: Benchmarks when training our MNIST model on the CPU.



(a) Forward pass (b) Backward pass (c) Loss computation (d) Communication

Figure A.2.: Benchmarks when training our UNet++ model on the CPU.

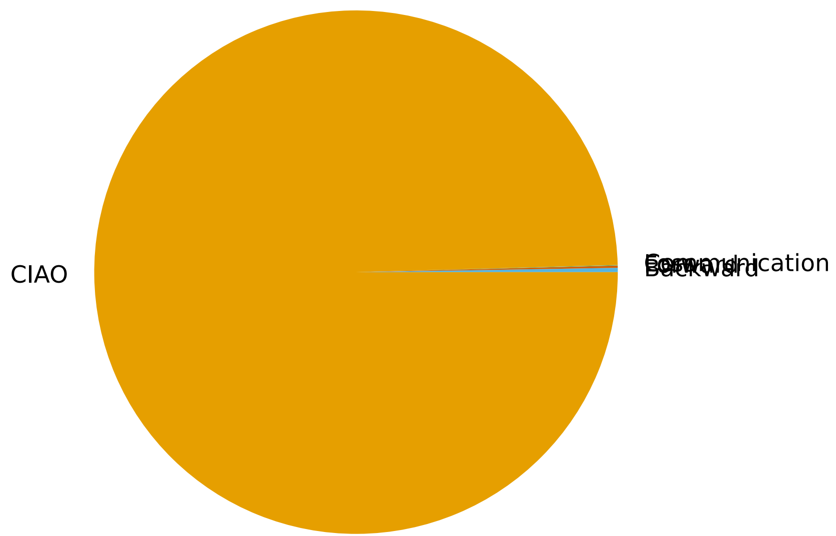


Figure A.3.

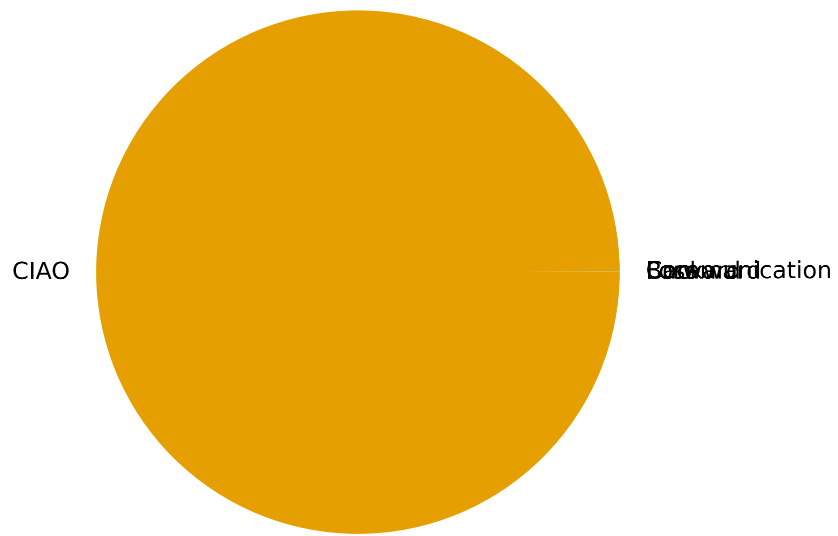


Figure A.4.

Bibliography

[1] examples/mnist/main.py at main - pytorch/examples. https://github.com/pytorch/examples/blob/main/mnist/main.py. Accessed: 2025-11-20.

[2] Stoffmenge. https://www.chemie.de/lexikon/Stoffmenge.html. Accessed: 2025-11-22.

[3] Information technology — programming languages — fortran. ISO/IEC Standard 1539-1:2018, International Organization for Standardization, Geneva, Switzerland, 2018.

[4] Programming languages — c++. ISO/IEC Standard 14882:2023, International Organization for Standardization, Geneva, Switzerland, 2023.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[6] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey, 2018.

[7] D. M. Beazley, D. Eddelbuettel, et al. Cffi: C foreign function interface for python. https://cffi.readthedocs.io/, 2025. Accessed 2025-12-10.

[8] C. M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, New York, 2006.

[9] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT)*, pages 177–186. Springer, 2010.

[10] CERFACS (PhyDLL developers). PhyDLL: Physics deep learning coupler (version 0.1.0). https://gitlab.com/cerfacs/phydll, 2025. [Online; accessed 2025-12-10].

[11] S.-K. Chao, Y. Xing, Z. Wang, and G. Cheng. Vgg16 dataset, dec 2024.

[12] D. Cireşan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification, 2012.

[13] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12):3207–3220, Dec. 2010.

[14] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik. Emnist: an extension of mnist to handwritten letters, 2017.

[15] Y. Collet. xxhash: Extremely fast non-cryptographic hash algorithm. `https://github.com/Cyan4973/xxHash`, 2025. Accessed: 2025-12-11.

[16] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[17] E. G. Dada, J. S. Bassi, H. Chiroma, S. M. Abdulhamid, A. O. Adetunmbi, and O. E. Ajibuwa. Machine learning for email spam filtering: review, approaches and open research problems. *Heliyon*, 5(6):e01802, 2019.

[18] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. a. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng. Large scale distributed deep networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[19] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[20] 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*, pages 248–255, 2009.

[21] O. Desjardins, G. Blanquart, G. Balarac, and H. Pitsch. High order conservative finite difference scheme for variable density low mach number turbulent flows. *Journal of Computational Physics*, 227:7125–7159, 07 2008.

[22] O. R. developers. Onnx runtime. `https://onnxruntime.ai/`, 2021. Version: x.y.z.

[23] S. Dymchenko, A. Purandare, and B. Raffin. Melissadl x breed: Towards data-efficient on-line supervised training of multi-parametric surrogates with active learning. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 32–40, 2024.

[24] et al. Revisiting model scaling with a u-net benchmark for 3d medical image segmentation. *Scientific Reports*, 15:15617, 2025.

[25] T. Feng, W. Li, D. Zhu, H. Yuan, W. Zheng, D. Zhang, and J. Tang. Zeroflow: Overcoming catastrophic forgetting is easier than you think, 2025.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[27] O. Gicquel, N. Darabiha, and D. Thévenin. Laminar premixed hydrogen/air counterflow flame simulations using flame prolongation of ildm with differential diffusion. In *Proceedings of the Combustion Institute*, volume 28, pages 1901–1908, 2000.

[28] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[29] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[30] T. Hilgers. Coupling turbulent boundary layer flow simulation with a transformer, 2026.

[31] Z. Hu, S. Shen, T. Bonato, S. Jeaugey, C. Alexander, E. Spada, J. Dinan, J. Hammond, and T. Hoefler. Demystifying nccl: An in-depth analysis of gpu communication protocols and algorithms, 2025.

[32] L. Huang, J. Qin, Y. Zhou, F. Zhu, L. Liu, and L. Shao. Normalization techniques in training dnns: Methodology, analysis and application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(8):10173–10196, 2023. Originally arXiv preprint arXiv:2009.12836 (2020).

[33] Intel Corporation. oneccl: oneapi collective communications library. `https://github.com/oneapi-src/oneCCL`, 2025. Accessed 2025-12-10.

[34] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[35] S. Kato and K. Hotta. Mse loss with outlying label for imbalanced classification, 2021.

[36] D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 25. Curran Associates, Inc., 2012.

[38] A. Laila and M. Kern. A global method for coupling transport with chemistry in heterogeneous porous media. *Applied–format reactive transport modelling (preprint / arXiv)*, 2009. arXiv preprint arXiv:0912.3867.

[39] C. Lameter. Numa (non-uniform memory access): An overview. *Communications of the ACM*, 56(7):59–67, 2013.

[40] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[41] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–48. Springer, 1998.

[42] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In G. Montavon, G. B. Orr, and K.-R. Müller, editors, *Neural Networks: Tricks of the Trade*, pages 9–48. Springer, 2012.

[43] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shearer, and B.-Y. Su. Scalable distributed dnn training using commodity gpu servers. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, pages 1177–1185. PMLR, 2014.

[44] N. Liesch. The bmp file format.

[45] Y. Liu, J. Cao, C. Liu, K. Ding, and L. Jin. Datasets for large language models: A comprehensive survey, 2024.

[46] A. Mao, M. Mohri, and Y. Zhong. Cross-entropy loss functions: Theoretical analysis and applications, 2023.

[47] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data, 2023.

[48] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 5.0*, June 2025.

[49] L. T. Meyer, M. Schouler, R. A. Caulk, A. Ribes, and B. Raffin. High throughput training of deep surrogates from large ensemble runs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[50] J. Montiel, M. Halford, S. M. Mastelini, G. Bolmier, R. Sourty, R. Vaysse, A. Zouitine, H. M. Gomes, J. Read, T. Abdessalem, et al. River: machine learning for streaming data in python. *Journal of Machine Learning Research*, 22:1–8, 2021.

[51] J. Montiel, J. Read, A. Bifet, and T. Abdessalem. Scikit-multiflow: A multi-output streaming framework. *Journal of Machine Learning Research*, 19(72):1–5, 2018.

[52] NVIDIA Corporation. Nccl: Nvidia collective communications library. `https://developer.nvidia.com/nccl`, 2025. Version 2.x, accessed 2025-12-10.

[53] OpenAI. Gpt-5, 2024. Large Language Model developed by OpenAI.

[54] F. Orland, L. Nista, N. Kocher, J. Vanvinckenroye, H. Pitsch, and C. Terboven. Efficient and scalable aixeleration of reactive cfd solvers coupled with deep learning inference on heterogeneous architectures. In *Proceedings of the 2025 International Conference on High Performance Computing in Asia-Pacific Region Workshops*, HPC Asia '25 Workshops, page 45–57, New York, NY, USA, 2025. Association for Computing Machinery.

[55] K. O'Shea and R. Nash. An introduction to convolutional neural networks, 2015.

[56] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks, 2013.

[57] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[58] L. Perez and J. Wang. The effectiveness of data augmentation in image classification using deep learning, 2017.

[59] T. Poinsot and D. Veynante. *Theoretical and Numerical Combustion.* R.T. Edwards, Inc., 2 edition, 2005.

[60] PyTorch Team. Pytorch examples: Mnist. `https://github.com/pytorch/examples/blob/main/mnist/main.py`, 2024. Accessed: 2025-12-10.

[61] J. D. Regele, E. Knudsen, H. Pitsch, and G. Blanquart. A two-equation model for non-unity lewis number differential diffusion in lean premixed laminar flames. *Combustion and Flame*, 160(2):240–250, 2013.

[62] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In B. Roy and W. Meier, editors, *Fast Software Encryption*, pages 371–388, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[63] Y. Roh, G. Heo, and S. E. Whang. A survey on data collection for machine learning: A big data - ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*, 33(4):1328–1347, 2021.

[64] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[65] S. Roth and A. Stahl. *Mechanik und Wärmelehre : Experimentalphysik – anschaulich erklärt*. SpringerLink. Springer Spektrum, Berlin, 2016.

[66] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, pages 318–362. MIT Press, Cambridge, MA, 1986.

[67] RWTH Aachen University. Claix-2023 high performance computing cluster. `https://www.itc.rwth-aachen.de`, 2023. High-performance computing cluster at RWTH Aachen University, Germany.

[68] S. Salman and X. Liu. Overfitting mechanism and avoidance in deep neural networks, 2019.

[69] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[70] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[71] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision, 2015.

[72] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2nd edition, 2007. Classic reference for the client–server model.

[73] J. V. Vanvinckenroye. A Posteriori Untersuchung von CNNs zur Modellierung von Wasserstoffverbrennung. Masterarbeit, RWTH Aachen University, Aachen, 2024. Veröffentlicht auf dem Publikationsserver der RWTH Aachen University; Masterarbeit, RWTH Aachen University, 2024.

[74] J. von Neumann. Various techniques used in connection with random digits. *National Bureau of Standards Applied Mathematics Series*, 12:36–38, 1951.

[75] T. Wen et al. Species transport equation in multicomponent systems. *Applied Energy*, 261:114402, 2020.

[76] G. Xu, X. Wang, X. Wu, X. Leng, and Y. Xu. Development of residual learning in deep neural networks for computer vision: A survey. *Engineering Applications of Artificial Intelligence*, 142:109890, Feb. 2025.

[77] D. Zha, Z. P. Bhat, K.-H. Lai, F. Yang, Z. Jiang, S. Zhong, and X. Hu. Data-centric artificial intelligence: A survey, 2023.

LLM Usage

[78] OpenAI GPT 4o. Chatgpt prompt: "please generate a tikzpicture that visualizes the compute graph of the following model" attached with the mnist model. ChatGPT conversation, 2025. Accessed: 2025-12-05.

[79] Mistral 7B. Prompt: Generate a tikz image for an overview of melissadl., 2023.

[80] Mistral 7B. Prompt: Generate a tikz image for the communicator design with each gpu node being combiend with at least one simulation process. the training processes interchange gradients using their own communicator., 2023.

[81] Mistral 7B. Prompt: Generate a tikz image of backpropagation, 2023.

[82] Mistral 7B. Prompt: Generate a tikz image of the traditional unet, 2023.

[83] GPT-5. Prompt : Generate a tikz picture of the following model (...), 2024. Large Language Model developed by OpenAI.

[84] GPT-5. Prompt : Generate a tikz picture of the steps taken for model training, 2024. Large Language Model developed by OpenAI.