MLOps Project

1. Introduction

This project focuses on MLOps, as covered in the MLOps course during the Spring 2025 semester of the Bachelors of Engineering in Artificial Intelligence Engineering program. The project is structured into sections that align with the corresponding lectures in the course. Each section begins with theoretical insights to emphasize the importance of the respective stage in a MLOps pipeline, followed by a description of how the theory was applied in practice. However, due to technical limitations, not every component could be fully implemented. Where implementation was not feasible, reflections are provided on the limitations and expected outcomes had those steps been realized. Compute resources provided by AILAB at the University of Aalborg were used, unless stated otherwise. As part of documentation, a GitHub repository was set up for this MLOps project, which can be found here.. A static ZIP-version of the repository is also provided in the hand-in. The source code for this project was made in collaboration with XXi
Data and binary files (e.g., model weights) were ignored in the GitHub, hence why they may not be present. Every reference with *.py* refers to a Python script, which can be found in the repository. In accordance to the rules of the Dept. of Architecture, Design, and Media Technology, I wish to clarify that Generative AI has been used as support for grammatical revision, which is further detailed in Appendix D.

1.1. Development Paradigms and Differences

MLOps is a development paradigm for Machine Learning (ML) systems that integrates principles from ML, traditional DevOps, and data engineering [1], [2]. Its primary goal is to enable efficient development, deployment, and maintenance of ML systems in production environments. Specifically, it is comprised of a set of practices that manage the entire life cycle of ML models, from data extraction and model training to deployment and monitoring. Unlike traditional approaches where only the model is deployed as an API, MLOps emphasizes automating the full ML pipeline, including continuous training and redeployment of models.

In software development, three common paradigms are the Waterfall model, Agile, and DevOps. MLOps builds upon these but introduces additional considerations specific to ML systems.

The Waterfall model follows a linear, step-by-step process, typically formulated as: $\textit{Requirements} \rightarrow \textit{Design} \rightarrow \textit{Development} \rightarrow \textit{Testing} \rightarrow \textit{Deployment} \rightarrow \textit{Maintenance}$. This sequential approach does not easily accommodate for changes once a stage is completed. For example, revising and modifying the requirements after the design or development phase often necessitates restarting the entire process [3].

In contrast, the Agile method adopts an iterative and progressive approach [1]. Agile emphasizes involvement of end-users, enabling continuous feedback and adaption. After each iteration, feedback is gathered, requirements are adjusted, and improvements are implemented, resulting in a cyclic, user-centered development loop.

DevOps builds upon Agile by enhancing collaboration between development and operations teams. It introduces a dual-cycle process, integrating automation practices such as Continuous Integration (CI) and Continuous Delivery/Deployment (CD). CI involves automatic testing and validation of code changes, while CD enables the automated deployment of these changes to production. Together, CI/CD pipelines ensure that software updates are tested, built, and deployed reliably whenever the codebase is changed.

Finally, MLOps extends DevOps to accommodate for the unique demands of ML systems. Unlike traditional software, ML systems evolve with new data, requiring Continuous Training (CT) to keep models accurate over time.

MLOps also introduces practices for automating model retraining and redeployment. Additionally, it addresses ML-specific challenges, such as dataset and model versioning, performance monitoring, and handling concept drift or data drift; concepts that are not typically present in traditional software development workflows.

1.2. Selected Project

To implement an MLOps pipeline, it must be built upon an ML or Deep Learning (DL) method. For this project, the selected method is a neural network that classifies car models based on input images; a model originally developed for a mini-project in a DL course.

The model architecture is based on ResNet50 with a Multilayer Perceptron (MLP) as the classification head. A pre-trained ResNet50 model was acquired from PyTorch's *model* class [4]. Training and testing scripts were developed to serve as the foundation for the MLOps project. The method was trained on the Stanford car Dataset, which contains 16,185 images (split 50/50 into train and test subsets) of 196 unique car models [5]. During training, the pre-trained weights of the ResNet50 were frozen, and only the MLP network was trained using cross entropy loss, see *train.py*. The model weights corresponding to the highest test accuracy were saved. Training and validation loss plateaued after approximately three epochs, which was tracked using Weights & Biases (WandB), achieving an accuracy of approximately 78% and a test loss of one, see Figure 1

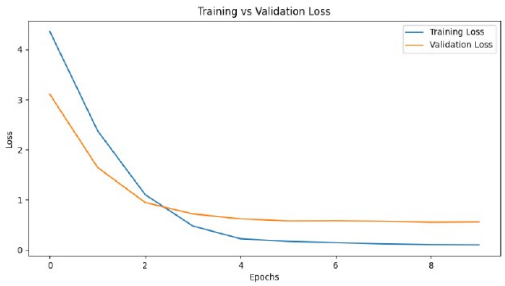


Figure 1. Training and validation loss per epoch. Notice how the validation loss plateaus around epoch three.

In accordance with the first lecture's instructions, a GitHub repository was set up for version control, and a *requirements.txt* file was created to define the necessary dependencies. Additionally, Data Version Control (DVC) in conjunction with Google Cloud storage was implemented for dataset and model versioning. DVC was chosen due to its Git-like interface, which aligned with out existing experience [6]. DVC creates .dvc pointer files that are version-controlled with Git. The actual files are pushed to Google Cloud, which acts as the remote storage. This allowed

```
outs:
- hash: md5
  path: data
  md5: b15d364e7db5c74c236c2f8d005ef01c.dir
  size: 1990113876
  nfiles: 16188
```

Figure 2. DVC file for data versioning.

us to push and pull different data/model versions from the Google Cloud storage. Finally, a YAML configuration file (*train_config.yaml*) was introduced to manage training hyperparameters and improve reproducibility. However, after using Google Cloud storage for a brief period, our free credits were spent, leading to us deactivating DVC and Google Cloud. To prove exercise completion, a screenshot is provided on the .dvc file for data versioning, see Figure 2. A similar file can be for model versioning (see *best_model.pth.dvc*).

1.3. Foreseen Challenges

While the ideal end-goal of this project is displaying a functional MLOps pipeline for image classification, several challenges are anticipated in extending and maintaining the system:

- **Development Consistency**: Ensuring identical environments across local machines of each contributor, CI/CD pipelines, and production may be difficult due to dependency drift and/or hardware variations.
- **Reproducibility**: Despite using DVC and YAML configuration files, full reproducibility may be hindered by non-deterministic GPU operations and manual overrides that are not captured in documentation or version control. Given that this is our first attempt at implementing an MLOps pipeline, creating robust documentation and versioning practices may be challenging and mistakes are likely to happen.
- **Monitoring**: The system currently lacks monitoring tools for tracking, for example, inference latency, failure rates, or performance drift, which are essential for long-term reliability. However, implementing these tools may cause dependency problems with our existing dependencies.
- **Maintenance**: As our model was trained on images of older car models with different image characteristics, the model can quickly become outdated. Mechanisms to detect and handle concept or data drift must therefore be considered.
- **Automation Complexity**: While the end-goal is automating retraining and potentially deployment, it requires defining clear triggers and robust validation

logic to ensure model quality is preserved. Furthermore, since compute resources from AILAB are utilized, it may be difficult to automatically initialize a retraining job due to it requiring VPN access and user verification.

Addressing these challenges can be critical for scaling the system from a course project to a real-world, production-ready AI solution. However, due to budget constraints and practical reasons, the practical goal of this project is primarily demonstrating a proof-of-concept MLOps pipeline, rather than delivering a fully automated, production-ready MLOps pipeline.

2. Continuous ML

With the selected project and its challenges outlined, the implementation of a continuous ML pipeline can now be addressed.

In DevOps, multiple branches are used to separate development activities from the production-ready *main* branch, preventing faulty code from being deployed [2]. MLOps adopts the same principle, as errors in production ML systems can lead to downtime or unreliable predictions. In this project, two separate branches were created: one for unit test development and one for general feature development, as shown in Figure 3. Additionally, branch protection was activated on the main branch to avoid force pushes, mitigating the risk of unwanted repository behavior or merging buggy code into the protected branch [7].

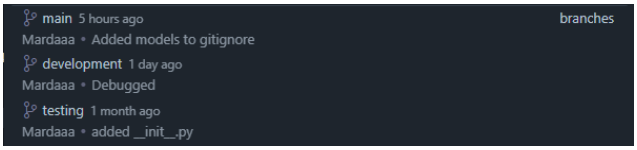


Figure 3. Screenshot of the branches. Besides the main-branch, development and testing branch were created.

To catch simple issues early, local checks were implemented using pre-commit hooks [8]. These hooks run predefined steps before a commit is accepted, functioning similarly to unit tests by canceling commits if any checks fail. Furthermore, the checks should be fast to encourage frequent commits. To manage and maintain pre-commit hooks, we adopted the *pre-commit* framework and followed the official guideline, see *.pre-commit-config.yaml* [9]. The configured hooks included: 1. removing whitespace at the end of lines in a script, 2. preventing inconsistent end-of-file formatting (i.e., by ensuring that files end with a single newline), 3. validating YAML files and ensure they have correct syntax, and 4. preventing large files (default setting is 500 kb) from being committed, which can be useful for avoiding accidental commits of large binary files.

While pre-commit hooks catch formatting and syntax issues, they do not verify the actual functionality of the codebase. During development, functions may be restructured or modified, and without proper testing, such changes may introduce unintended errors being pushed to production [10]. To safeguard against this, unit testing is essential, as it ensures that individual functions behave as intended even after modifications [11]. For this project, unit tests were implemented using the *PyTest* library. Due to technical issues with implementing DVC and Google Cloud storage, not all group members had access to the dataset at the time of testing. Therefore, we focused solely on testing the inference script, which was accessible and did not require direct interaction with the full dataset. Ideally, unit testing should extrend to all components, including the training pipeline (e.g., veryfing that data loads correctly or that loss decreases during training).

The original *inference.py* script from the mini-project consisted of a single large main function handling tasks like model loading, image preprocessing, and prediction. To make it testable, the script was refactored into modular functions and saved as *src/new_inference.py*. Corresponding tests were implemented in *tests/test_inference.py*. These tests covered functions for image loading and preprocessing, model prediction, and class name retrieval. The tests were run using PyTest, and as shown in Figure 4, all five tests passed successfully. Note that the warnings seen are related to the Conda environment and not the tests themselves.

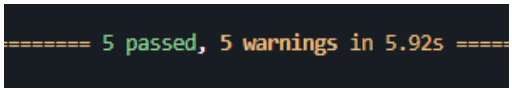


Figure 4. Unit test results, showing that all five tests passed.

Assessing the quality of unit tests can be challenging, but ensuring sufficient test coverage is crucial to avoid unintended failures in production [10]. One way to evaluate this is through code coverage, which measures the percentage of the codebase that is executed during testing. While 100% coverage does not guarantee bug-free code, it indicates that all lines of code are at least executed during tests. For this purpose, we used the *coverage.py* package, which integrates well with PyTests and generates detailed coverage reports [12]. Figure 5 shows the coverage report our unit tests, which achieved 63% coverage. After investigating, the uncovered lines corresponded to the model-loading function, the main logic-wrapper, and the script entry-point.

These could be tested in a similar fashion as the already tested components. However, given that the goal was to build a learning-focused and practical (not perfect) MLOps pipeline, we prioritized practical insights over 100% coverage.

```
Name                    Stmts   Miss  Cover   Missing
-----------------------------------------------------
model.py                   17     11    35%   8-19, 31-33
src\new_inference.py       46     17    63%   17-21, 51-64, 67
tests\__init__.py           0      0   100%
tests\test_inference.py    49      0   100%
train.py                   87     70    20%   15-17, 20-24, 29-47, 50-70, 73-157, 161
-----------------------------------------------------
TOTAL                     199     98    51%
```

Figure 5. Code coverage of the unit tests. The coverage of the inference script is 63%. Note, the missing coverage of the script is the model-loading function, the main-function, and the call of the main-function.

To avoid the risk of forgetting to manually run tests, we automated this process using GitHub Actions, which is tightly integrated with GitHub and easy to configure for CI/CD workflows [13]. A limitation of GitHub Actions is the limit on automation minutes (2000 free automation minutes per month) and less flexibility compared to self-hosted alternatives. However, for this educational project, GitHub Actions was deemed sufficient. Per the official guidelines of GitHub Actions, a configuration file was created that 1) installed dependencies and 2) ran unit tests and reported the results for each push to the main-branch. This configuration can be found at *.github/workflows/unittests.yml*. Figure 6 shows a successful execution of this automated test execution after a push to the main branch. At the current moment in time, subsequent workflows fail under the Action-panel on GitHub due to free credits being spent.

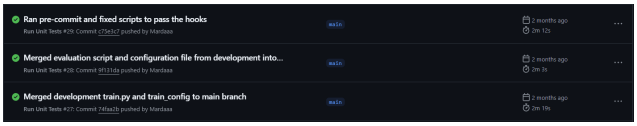


Figure 6. Screenshot of the GitHub Action-page, where it can be seen the unit test were run automatically after pushes.

Per the exercise requirements, CT had to be implemented when all unit tests passed to ensure models remained accurate over time (as described in Section 1). After each successful training session, the newly trained model was to be registered in a model registry, enabling proper versioning and backtracking. To automate this process, we followed AILAB's official guidelines and created a YAML workflow file (*.github/workflows/ci.yaml*) designed to trigger a test job on AILAB, as shown in Figure 7 [14]. Unfortunately, despite numerous attempts, we were unable to successfully establish a connection to AILAB via

```yaml
name: AI-LAB test job
on:
  push:
    branches:
      - main
jobs:
  test-hpc:
    runs-on: self-hosted
    steps:
      - name: Check out repo
        uses: actions/checkout@v3
      - name: AI-LAB hostname info
        run: |
          sbatch --output=hostname_output.txt --wrap="hostname"
```

Figure 7. Screenshot of the YAML-file that was supposed to execute a job on AILAB.

GitHub Actions. Even with assistance from CLAAUDIA (AILAB's IT support), we could not overcome the technical barriers, specifically the inability to connect through AAU's SSH gateway. While we could have manually trained and evaluated the model locally, we intentionally chose not to bypass the CI/CD objective of the exercise, as this would defeat the purpose of learning proper automation practices. If SSH access to AILAB had been established, the automation process would mirror the unit test pipeline: set up the conda environment, execute *train.py* and *evaluation.py* with the newly trained model. Assuming it was possible, CT would have been used to ensure that the ML model remains accurate and reliable as data evolves over time (further covered in Section 6), thus allowing it to adapt to changing real-world conditions efficiently. That is, retraining the model when triggered by, for example, new data, changes to model architecture, or performance.

Figure 8 provides an overview of the implemented MLOps pipeline. Components we failed to implement due to technical limitations and potential future automations are marked with red.

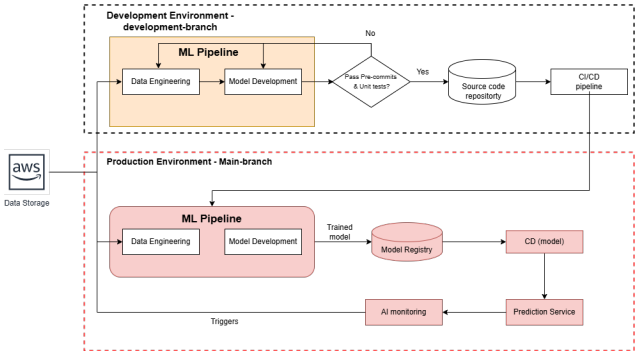


Figure 8. Flowchart of the implemented Continuous ML pipeline we implemented. Red means that the component was not implemented due to technical limitations or had yet to be implemented.

As seen in Figure 8, the process starts by fetching data from the Google Cloud storage using DVC. Note, in the figure it is represented by AWS S3 storage, because we initially

used AWS S3. However, we transitioned over to Google Cloud storage, as our free AWS S3 credits were spent over the span of a couple of days. Ultimately, the data is fetched from cloud storage regardless of platform. Next the data is passed to the Development Environment (or the *development*-branch), where the ML pipeline is executed (i.e., data engineering and model development, such as modeling and evaluation). If the source code passes pre-hooks and unit tests, the source code is pushed to the source code repository (i.e., merged with the main branch). Then, the ML pipeline was supposed to trigger automatic training (and evaluation), resulting in a trained model. Afterwards, the trained model was supposed to be added to a model registry, get deployed and served for predictions. Again, we were unable to actually implement these steps due to technical limitations. For future automation, the goal is to implement ML monitoring and if the model's performance degrades below a certain threshold, retraining can be triggered to ensure it stays accurate over time.

3. Scalable Training

Over recent years, DL methods have become increasingly reliant on compute power for training, driven by the growth in model sizes [15]. For example, modern transformer models demand up to 275 times more compute power for training compares to those from 2017. Additionally, the memory footprint of modern DL models has grown to the point where they barely fit into the memory of a single GPU. As this trend continues, scaling is necessitated.

3.1. Scaling Laws

Scaling refers to the process of increasing one or more of the following variables: compute resources (GPUs, compute nodes, etc.), dataset size, and model parameters [15], [16]. Expanding these variables provides several benefits, such as improved model performance and faster training times. A key question is how much these variables should be increased to improve model performance, specifically to lower the test loss. Assuming the model design is reasonably good, *Scaling Laws* offer a way to estimate the test loss when increasing one of the three variables [16]. The scaling laws for compute (D), dataset size (D), and parameters (N) are as follows:

$$L(C) \approx \left(\frac{C_c}{C}\right)^{\alpha_C}; \alpha_C \sim 0.05, C_c \sim 3.1 \times 10^8 \quad (1)$$

$$L(D) \approx \left(\frac{D_c}{D}\right)^{\alpha_D}; \alpha_D \sim 0.095, D_c \sim 5.4 \times 10^{13} \quad (2)$$

$$L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha_N}; \alpha_N \sim 0.076, N_c \sim 8.8 \times 10^{13} \quad (3)$$

Where *C, D,* and *N* represent compute (measured in PF-days), dataset size (measured in tokens or samples), and the number of model parameters, respectively [16]. Additionally, one PF-day is equivalent to 8.64×10^{19} Floating Point Operations (FLOPs). Using these equations, the theoretical scaling needed to halve the current test loss for the ResNet50 classifier can be computed. Note that the constants provided for Eq. 1 to Eq. 3 are based on Large Language Models (LLMs). Since the constants for the ResNet50 classifier are unknown, the LLM-constants were used for this calculation. Recall from Figure 1 that the validation loss was approximately 1 after completing training the ResNet50 model. By applying the LLM-constants, the required scaling can be computed, see Figure 9.

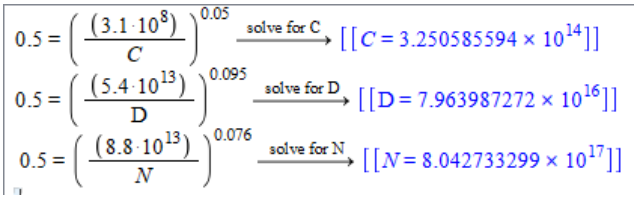


Figure 9. Computation of the scaling required for C, D, and N, respectively, in order to halve the validation loss.

As illustrated in Figure 9, a significant amount of scaling is required to reduce the test loss by half. For example, the training dataset size needs to increase from 8,144 samples to 7.96×10^{16} samples, indicating that exponential growth quickly becomes impractical. However, the LLM constants were used for the computation, and using the appropriate constants for the ResNet50 classifier may yield a more realistic outcome. The main takeaway, however, is that typically a 10,000x increase in size is needed to halve the validation loss [15], [16]. The question then becomes whether it is worth scaling to achieve such a reduction in test loss. Despite the vast computational resources demand for larger models, they remain preferable because they learn faster than smaller models; they are more sample-efficient and can learn tasks faster compared to smaller models [15]. Therefore, scaling the model size continues to be relevant. Nevertheless, it is unwise to increase the number of parameters indefinitely without considering the impact on training time. As the number of parameters grows, so does the training time. This highlights the importance of parallelization, where training computations are distributed across multiple GPUs.

3.2. Parallelization

The theoretical speed-up in training-time when distributing the workload across multiple GPUs can be estimated using Amdahl's Law or Gustafson's Law [17], [15]. Amhdahl's

Law describes the theoretical speed-up of a task when multiple processors (e.g., GPUs) are used, assuming that a fraction of the task can be parallelized. In DL, several tasks can be parallelized, while others remain sequential. Examples of parallelizable tasks include forward pass, backward pass, and weight updates, which primarily involve matrix multiplications and gradient computations. These tasks are highly parallelizable due to the Single Instruction, Multiple Data (SIMD) nature of vector processors [18]. On the other hand, sequential tasks include model synchronization (in Distributed Training) and miscellaneous operations, such as logging and control flow logic [15]. However, Amdahl's Law is limited because it assumes that the problem size (e.g., number of operations, training steps, etc.) remains fixed regardless of the number of processors used [17]. In practice, the problem size often increases as the number of GPUs are added, which is accounted for by Gustafson's Law. Thus, Gustafson's Law is more appropriate for estimating speed-up in DL tasks. Gustafson's Law is given by:

$$S(P) = P - \alpha(P - 1) \tag{4}$$

Where S denotes the speed-up, P represents the number of processors, and α is the non-parallelizable fraction of the task [17]. For different fractions of non-parallelizable tasks and number of processors, the estimated speed-up can be seen in Figure 10. Note that different values for the non-parallelizable fraction were used, as the exact fraction for ResNet50 could not be found in literature. Consequently, the figure should be considered as a reference. In retrospect, PyTorch's *Profiler* package could have been employed to obtain an Execution Summary of the training script [19]. While the profiler does not explicitly indicate whether a task is parallelizable, it provides a breakdown of processes (e.g., DataLoader activity and kernel operations) along with their respective utilization percentages. This breakdown could be used to infer the non-parallelizable fraction of the training session. However, I was unaware of this at the time of calculations.
As seen in Figure 10, two processors can speed up the training process by a factor of ~ 2, while eight processors can provide a speed-up of $\sim 6.3 - 7.6$ times. This highlights the significant impact of parallelization on training speed, giving motivation for implementing it in this project. Next is determining which parallelization method to use.

There are two primary scaling paradigms, or parallelization methods, that can be employed: data parallelism and model parallelism [20], [15]. In **data parallelism**, the data is divided into n partitions, where n is the total number of available worker nodes in the compute cluster. The full model is then copied onto each worker node, assuming it can fit on a single node, and each node processes a distinct

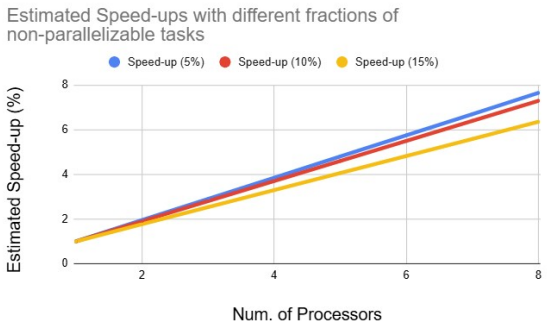


Figure 10. Estimated speed-up (%) with different number of processors and fractions of non-parallelizable tasks. Blue, red, and yellow represent a non-parallelizable fraction of 5%, 10%, and 15%, respectively.

micro-batch of the data. During training, the worker nodes exchange information and synchronize their gradient updates using the all-reduce algorithm.

In contrast, **model parallelism** partitions the model itself, rather than the data. This can be done either horizontally, known as tensor parallelism, or vertically, which is known as pipeline parallelism, so that different portions of the model are processed concurrently [20]. In this paradigm, all nodes operate on the same dataset. As a general rule of thumb, model parallelism is only employed when the model cannot fit on a single worker node [15]. Otherwise, data parallelism is generally preferred due to its ease of implementation relative to the performance gains. Regardless of the chosen scaling paradigm, communication between worker nodes is essential [15]. However, not all communication interfaces offer the same performance. For intra-node communication, NVLink provides optimal bandwidth and latency. For inter-node communication, InfiniBand is generally preferred. Unfortunately, AILAB lacks both NVLink and InfiniBand support. Consequently, the speed-ups observed in the following scaling experiments may have been more pronounced if these communication technologies had been available. Note, in the following experiments, providing raw screenshots of the results are infeasible. Therefore, to prove exercise completion, screenshots of GPU utilization are presented in Appendix A.

3.2.1 Scaling Across Two GPUs

Based on the considerations outlined in previous section, we opted for data parallelism for this project, as the ResNet50 architecture easily fitted within the memory limits of a single GPU. We implemented data parallelism using PyTorch's Distributed Data Parallel (DDP) package, which was chosen for its seamless integration with our

existing scripts leveraging the PyTorch framework. The DDP package was implemented per the instructions of the official documentation, see *scalable training/train ddp.py*, lines 88-98. Scaling was applied across two GPUs on the same compute node. To execute these training experiments, *Torchrun* was used [21].

During the experiments, 24 GB of memory was used when training on a single GPU, while this increased to 48 GB when two GPUs were used. In both cases, the ResNet50 architecture was trained for three epochs with a batch size of 16. The impact of using two GPUs on the training time is presented in Table 1, along with the percentage difference, which was computed using the following formula:

$$difference\% = \frac{|Value_1 - Value_2|}{\frac{(Value_1 + Value_2)}{2}} \times 100 \quad (5)$$

where $Value_1$ is the single-GPU training time and $Value_2$ is the new training time [22].

Speed-up in Training Time (Same Compute Node)		
Configuration	**Batch size = 16**	**Difference %**
No DDP	479.57 s	–
With DDP	294.38 s	47.86%

Table 1. Effect of data parallelism on the training times. Two GPUs were used from the same compute node. Notice how the execution time is significantly faster when using data parallelism with two GPUs.

As shown in Table 1, the use of data parallelism with two GPUs resulted in a training speed-up of approximately $1.7x$, corresponding to a 47.86% reduction in training time. It is worth reiterating that the system lacked NVLink, so this performance improvement could potentially have been greater under better hardware conditions. However, as learned with Gustafson's Law, which suggests that the problem size increases with the number of processing units, it is likely that the chosen batch size of 16 may have been suboptimal. A small batch size can lead to underutilization of available GPU resources, which is referred to as GPU starvation or underutilization, thereby limiting the achievable speed-up; this is further explored in Section 4 [15]. To investigate GPU utilization with this batch size, I launched a separate singularity shell on the same compute node using the *–nodelist*-argument. I then monitored GPU usage with the *"nvidia-smi"* command-line tool. The GPU utilization is presented in Figure 11.
As seen in Figure 11, the GPU utilization reached 97% and 98% for the two GPUs, respectively, indicating efficient usage of hardware resources. This high utilization suggests that the observed training speed-up is near-optimal for the given setup. Nonetheless, we further experimented varying

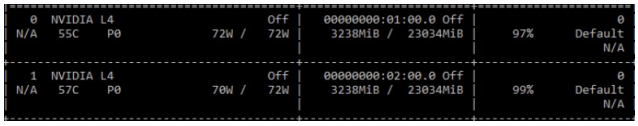


Figure 11. GPU utilization when running the *train ddp.py* script with a batch size of 16.

batch sizes to examine their effect on training time. The batch sizes tested were 8, 18, 20, 26, and 32. The resulting training times, along with their percentage difference compared to the baseline single-GPU setup, are presented in Table 2. The fastest observed training time is marked in **bold**. Screenshots of the GPU utilization for each configuration are provided in Appendix A.

Training Times with Different Batch Sizes		
Batch size	**Training Time (s)**	**Difference %**
8	311.58	42.47%
18	**280.23**	**52.47%**
20	281.80	51.95%
26	282.85	51.60%
32	284.62	51.02%

Table 2. Training times with different batch sizes. Two GPUs were used.

As seen in Table 2, the fastest training time was achieved with a batch size of 18, yielding a 52.47% reduction in training time compared to single-GPU training. Compared to the batch size of 16, there was a approximately a 14-second reduction. GPU utilization across these batch sizes (see Appendix A) was largely comparable to that of the batch size 16 configuration, except for the batch size of 8, where GPU utilization was noticeably lower. This lower utilization is likely the answer for the reduced speed-up observed with that configuration. Nonetheless, the utilization for all configurations were larger than 90%, and all configurations showed significant speed-up in training times.

Although not shown in the table, accuracy metrics were also tracked during the experiments. Training with a batch size of 16 (with DDP) yielded a validation accuracy of 78.7%, while increasing the batch size to 32 led to a decreased accuracy of 73.81% (screenshots are provided in Appendix A). Reflecting back on the experiment, this decline is likely contributed by the unchanged learning rate across experiments. Theoretically, when increasing the batch size by a factor of k, the learning rate should also be scaled by either k [23], or $\sqrt{k}$ [24]. Larger batch sizes produce more stable gradient estimates, making it feasible to use larger learning rates and potentially achieve faster convergence. However, using a small learning rate despite larger batch sizes may

lead to the optimizer to converge to a local minimum, due to reduced gradient noise. As such, if a larger learning rate was used, the accuracy would likely have been preserved. Unfortunately, I was not fully aware of this principle at the time of experimentation.

3.2.2 Scaling Across Two Compute Nodes

Rather than only scaling across two GPUs on the same compute node, we extended our experiments to a distributed setup spanning two compute nodes. Similar to before, this was done using *Torchrun*, as it enabled us to run the *train_ddp.py*-script with minimal configuration changes. For this experiment, two GPUs were used per compute node, and data parallelism was employed.

The setup proceeded as follows: two singularity shells were launched on AILAB, with each shell initialized on a separate compute node. The compute nodes were specified using the *–nodelist* argument. One node was designated as the master (assigned rank 0), and the other as the worker (assigned rank 1). To coordinate startup, the IP address of the master node was retrieved and passed to the *Torchrun* command on both nodes, ensuring that the worker node only began processing after the master had initialized communication. The effects of this implementation can be seen in Table 3. Since four GPUs were used, we tested two batch sizes: 30 and 60. The percentage difference in training time compared to the single-GPU baseline is also presented.

Speed-up in Training Time (Two Compute Nodes)		
Batch size	Training Time (s)	Difference %
30	652.06	-30.48%
60	301.41	45.62%

Table 3. Effect of data parallelism on the training times. Two GPUs were used for each compute node. Minus difference means that the training took longer than the baseline.

As seen in Table 3, when the batch size was 30, it took longer to train than the single-GPU baseline. Increasing the batch size to 60 resulted in a substantial reduction in training time (more than a factor of $2x$), indicating improved GPU utilization. As previously discussed, the expected speed-up may have been higher if Infiniband interconnects were available between the compute nodes. The absence of this may have led to the network communication latency becoming a bottleneck, thus limiting scalability.

3.3. Memory Optimization Strategies

In addition to parallelization for accelerating training, memory optimization strategies can also be utilized to reduce GPU memory consumption, allowing for larger batch sizes, improved hardware utilization, and thus, reduced training times [15]. Based on that, we implemented and experimented with two strategies: Automatic Mixed Precision (AMP) and Zero Redundancy Optimizer (ZeRO), due to their seamless integration with PyTorch [25], [26].

3.3.1 Automatic Mixed Precision

AMP enables the use of mixed-precision training by automatically selecting between half-precision (FP16) and full-precision (FP32) for each operation [25]. Since not all computations require full precision, this approach reduces memory usage and accelerates training. AMP was implemented following PyTorch's official guidelines (see lines 39-50 in *scalable_training/train_ddp_amp.py*) [27]. The experimental setup mirrored the configuration from Section 3.2.1, meaning two GPUs were used intra-node, and *Torchrun* was used to execute the experiment. The effects of AMP on the training time can be seen in Table 4, where the baseline refers to the training session without any scaling.

Setup	**Training Time (s)**	**Difference %**
Baseline	497.57	–
AMP	236.56	71.11%

Table 4. Effect of AMP on training time.

As observed in Table 4, AMP yielded a speed-up of approximately $2.1x$, corresponding to a reduction of 71.11%. Furthermore, this aligns with the estimations from Gustafson's Law (recall Figure 10), supporting the practical relevance of such scaling laws in GPU workloads. Furthermore, applying AMP yielded faster training times than data parallelism without AMP, proving that memory optimization can further decrease training times.

3.3.2 Zero Redundancy Optimizer

Unlike standard data parallelism, where each GPU holds a copy of the model, ZeRO reduces memory redundancy by partitioning model states (e.g., optimizer states, gradients, and parameters) across GPUs [26]. This minimizes communication overhead and allows for training larger models while maintaining efficient memory usage.

ZeRO was implemented following the official documentation (see *scalable_training/train_deepspeed.py* and *ds_config.json*) [28]. ZeRO supports multiple stages:

- Stage 1: Optimizer states are partitioned.
- Stage 2: This stage extends Stage 1 by also partitioning gradients.
- Stage 3: This stage also adds partitioning of model parameters.

We evaluated all three stages under the same experimental setup as in Section 3.2.1. Initially, a batch size of 16 and 18 were tested, since these yielded high GPU utilization in previous experiments. However, due to long execution times in later stages, this was increased to 32 in follow-up runs. The execution times of each stage can be seen in Table 5.

Stage	Batch size=16	Batch size=18	Batch size=32
1	288.00 s	260.88 s	**255.41 s**
2	292.98 s	261.61 s	**257.94 s**
3	607.18 s	614.09 s	**396.50 s**

Table 5. Effect of ZeRO on the training time in all three stages.

As shown in Table 5, increasing the batch size substantially improved execution times. While this initially suggested underutilization of the GPUs, *nvidia-smi* revealed a utilization of 100% for both GPUs at a batch size of 18 (see Appendix A); the utilization for a size of 16 was untested. This indicates that the slow training times in later stages are likely attributed to increased communication overhead rather than insufficient GPU workload. Specifically, the progressive increase in training time across ZeRO stages correlates with the extent of model state partitioning. For instance, in Stage 3, where optimizer states, gradients, and model parameters are all distributed across GPUs, inter-GPU communication demands rise. Due to the absence of NVLink, the setup likely encountered communication latency and bandwidth limitations, which may explain the training times. Even with full GPU utilization, the GPUs may have been "busy waiting" on data transfers rather than performing computations. Additionally, these experiments were executed using *Torchrun* instead of DeepSpeed's native launcher due to compatibility issues with our container setup on AILAB. This constraint may have further hindered performance, as DeepSpeed is optimized for its own execution pipeline [26]. Lastly, we encountered challenges running ZeRO with FP32, forcing us to half the precision to FP16. This change led to a slight reduction in accuracy; $\sim 68\%$ compared to $\sim 78\%$ in earlier experiments, see Appendix A.

Based on all these scaling experiments, it is evident that scaling across multiple processors, whether on the same node or different nodes, can yield significant speed-ups in training time, especially when using AMP in conjunction with data parallelism. With ZeRO, our results indicated that the training time was not reduced in Stage 3. However, in general, if NVLink and InfiniBand were available, faster speed-ups may have been observed.

4. Scalable Inference

As discussed in previous section, DL models have grown significantly in size in recent years. While this increase demands scalable training, it also calls for scalable inference (i.e., making model predictions efficiently once training is complete) [29]. To achieve faster inference, it is essential to ensure continuous and high GPU utilization [30]. Before exploring how this can be accomplished, three fundamental concepts must be defined:

- **Latency**: The time it takes to complete a single (e.g., the time required to perform inference on a single image) [31].
- **Throughput**: The number of opeprations (e.g., inferences or data packets) processed per unit of time, often measured per second [32].
- **Bandwidth**: The maximum rate at which data can be transferred (e.g., memory transfer from CPU to GPU) [33].

To reduce latency and improve throughput, the GPU must be kept busy in terms of both memory transfers and core utilization, as opposed to underutilized [30]. For memory efficiency, data should be continuously available in the GPU, minimizing transfers between CPU and GPU. For computational efficiency, the GPU's cores should be kept active and parallelized to achieve maximum throughput. While scaling compute power, dataset size, and model size often leads to improved model performance during training (as discussed in Section 3), the same does not necessarily hold true for inference time [29], [30]. Therefore, to balance model accuracy and inference time, it is common to apply model compression techniques (e.g., quantization) during deployment to accelerate inference while maintaining acceptable accuracy performances.

4.1. Model Compression

Model compression aims to shrink model size, speed up inference, lower computational costs, and increase energy efficiency [30]. Common techniques include:

1. Quantization.
2. Pruning.
3. Knowledge Distillation.

Quantization reduces memory usage and speeds up inference by converting model weights and activations into lower-precision formats (e.g., FP32 to INT8) [34]. This can reduce memory usage by $4x$ and increase throughput by $2 - 4x$ [30]. While it is relatively easy to implement and usually results in only a minor accuracy drop, careless quantization can result in a significant loss of accuracy.

Hence, it is important to find a mapping that minimizes the information loss through calibration.

Pruning removes less important weights or connections in the model (i.e., if they do not contribute much to the final prediction) [35]. However, the impact varies: some models tolerate pruning well, others experience significant drop in accuracy [30].

Knowledge distillation trains a smaller "student" model to mimic a larger "teacher" model by learning from the teacher's output logits [36], [30]. This often preserves accuracy well but is more complex to set up and tune.

In this project, quantization was chosen due to its balance between implementation difficulty and performance. Pruning is easier to implement but can cause a large drop in accuracy, while distillation is more reliable but harder to implement.

4.2. Quantization

Quantization can be applied either during training (QAT) or after training (PTQ) [30], [34]. QAT typically yields slightly better accuracies but requires retraining and is generally more complex to implement. In terms of inference time, however, there is no significant difference between QAT and PTQ. Since our models were already trained, we opted for PTQ to avoid the added complexity of retraining.

PTQ can implemented in two ways: dynamic and static. *Dynamic quantization* converts only weights ahead of time, while activations are quantized at runtime [34]. In contrast, *static quantization* uses a calibration dataset to quantize both weights and activations ahead of time. Static PTQ often delivers better performance and accuracy, but it is less flexible when data distributions vary. Our model was trained on images of older cars under varying lighting and background, leading to a diverse input distribution. Therefore, dynamic PTQ was the preferred choice, as it adapts to these variations on-the-fly rather than relying on fixed calibration ranges.

We implemented dynamic PTQ using PyTorch's *quantization* package, see *scalable_inference/quant.py* in the repository [37]. The model, originally in FP32, was converted to INT8 using the *quantize dynamic* function (lines 13-15 in *quant.py*), targeting only the weights of fully connected layers (i.e., *nn.Linear*). To our best knowledge, this limitation arises from PyTorch's current lack of dynamic quantization support for convolutional layers. To assess the impact, we compared the original FP32 model and the INT8 quantized model in terms of accuracy and average inference time measured across the test set. All benchmarking was done on CPU with a batch size of 16, as dynamic quantization in PyTorch is currently CPU-only. This implies that the inference time may be longer than what could be achieved on GPU-accelerated models. The effects of quantization is presented in Table 6. Upon further investigation, GPU quantization support exists in PyTorch's AO quantization [38]. However, as of 2024, these functions were prototypes, and it is unknown whether these have fully matured as of yet.

Model	Accuracy	Avg. Inference Time
Standard	71.09%	43.14 ms
INT8 Quantized	71.19%	41.81 ms

Table 6. Performance of standard model and dynamically quantized INT8 model.

As seen in Table 6, quantization yielded a slight accuracy improvemt (+0.1%) and a speed-up of approximately 3.13%. This is lower than typical expectations, where quantization can offer up to $4x$ speed-ups. A potential reason is that only the fully connected layers were quantized, while convolutional layers remained in FP32, which typically dominate computation time. Moreover, due to dynamic quantization's on-the-fly activation processing, it may have introduced overhead, especially since only a small portion of the model was quantized.

4.3. Batch Inference

In addition to model compression techniques, batch inference can be employed to increase GPU throughput [30], [39]. As the term suggests, instead of processing a single input sample at a time, a batch of, for example, eight images is processed concurrently. The general concept involves batching multiple input tensors from various requests, executing inference in a single forward pass, then un-batching and returning the respective predictions. This helps preventing GPU starvation. However, a drawback is increased latency per request, making it essential to consider and balance throughput and latency.

Based on the above, we implemented batch inference using FastAPI. FastAPI enables defining HTTP endpoints that can handle multiple concurrent requests, facilitating batch processing [30], [40]. The setup was deployed locally using FastAPI in conjunction with Uvicorn, enabling interaction via a locally hosted web browser, see *scalable_inference/batch_inference/app.py*. Due to technical constraints, we were unable to serve the PyTorch INT8-quantized model through FastAPI. As a workaround, the original FP32 model was converted to ONNX format and quantized to INT8 dynamically using ONNX' *quantization* package (see *batch_inference/convert_to_onnx.py*). We then defined an API endpoint that loaded images from a folder,

grouped them into batches of user-defined size, and performed inference using the quantized ONNX model. The endpoint returned the average latency (in ms) and throughput (images per second) for the specified batch. An example response is shown in Figure 12.

Figure 12. Screenshot of the FastAPI + Uvicorn interface. After submitting a batch of images, the application performed inference and returned latency and throughput metrics in JSON format.

To analyze the trade-off between latency and throughput, we measured both metrics across varying batch sizes: 1, 2, 4, 8, 16, and 32. The results are visualized in Figure 13, where the red annotations represent the respective batch size. The raw measurements can be found in Appendix B.

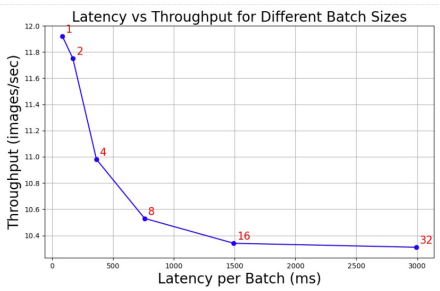


Figure 13. Throughput versus latency for different batch sizes. Each batch size is annotated in red. Notice the throughput is highest at a batch size of 1 and decreases with larger batches.

As shown in Figure 13, the highest throughput was observed with a batch size of 1. Increasing the batch size led to higher latency and lower throughput, which may seem counterintuitive given the typical advantages of batch inference. However, this behavior can be attributed to hardware limitations; we had difficulties implementing the FastAPI application on AILAB. Therefore, the experiment was conducted on a local machine without GPU acceleration via CUDA, due to outdated hardware. While CPUs are generally optimized for sequential executions, GPUs excel in parallel processing due to their SIMD architecture and large number of simple cores (Section 3.2). As such, increasing the batch size likely became a bottleneck on the CPU, reducing throughput instead of increasing it.

Under proper hardware circumstances, one would expect the opposite trend: small batch sizes would result in underutilized GPU resources (i.e., low throughput), whereas moderate batch sizes would increase throughput until a saturation point, beyond which the performance may degrade due to memory or compute constraints. In such scenario, it would be possible to empirically determine an optimal batch size balancing latency and throughput. Initially, we planned to use NVIDIA's Triton Inference Server, which offers dynamic batching to automatically optimize and balance this trade-off [41]. Unfortunately, due to persistent technical issues in pointing the Triton server to the model repository (leading to "404 Not Found" errors), we were unable to integrate Triton into the pipeline.

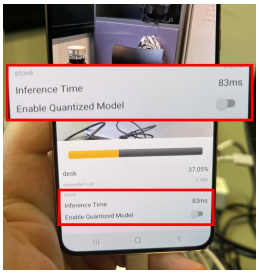
5. Deployment

Following the model optimization in Section 4, the next step in an MLOps pipeline is deployment. ML models can be deployed in three primary ways: cloud, edge, and self-hosted deployment [42], [43].

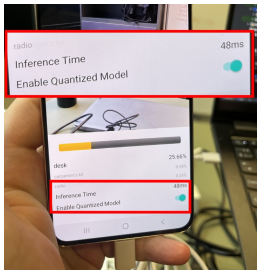
Cloud deployment provides a highly scalable infrastructure, offering vast amounts of computing resources provided over the internet [42], [43]. However, it comes with increased latency due to the physical distance between the deployed model and the data source, as well as less freedom and control over the hardware used. On the other hand, edge deployment moves computation closer to the data source by running the model on local hardware, such as smartphones or edge devices. This typically reduces latency but is constrainted by limited computational power. The third option, on-premise deployment, allows full control over hardware and data privacy but may be more complex to manage and scale [44].

Although this lecture was not directly tied to this project, we were tasked with deploying a separate image classification model on a smartphone, simulating a request from a "company". Specifically, the given model, provided in ONNX format with FP32 precision, was to be deployed on a Samsung Galaxy S21 (i.e., an example of edge deployment) with fast inference time as a key requirement. The phone was set up in developer mode to facilitate this.

Deployment was achieved using Microsoft's publicly available GitHub repository designed for deploying ONNX models onto Android devices via Android Studio [45]. As edge devices often have limited computing power, we first tested the inference speed of the FP32 model. As shown in Figure 14a, inference took approximately 83 milliseconds. To improve this, we applied model compression through quantization (as covered in Section 4), which leads to a small reduction in accuracy for better inference perfor-



(a) Inference time of FP32 model.



(b) Inference time of UINT8 model.

Figure 14. Inference times of the deployed image classification model on a Samsung S21 phone. The FP32 model and UINT8 model achieve an inference speed of 83 milliseconds and 48 milliseconds, respectively.

mance and lower memory usage. Among the compression techniques, quantization was particularly attractive due to its ease of implementation and often minimal impact on accuracy. Specifically, we quantized the model from FP32 to UINT8 precision. UINT8 is similar to INT8, except it only stores positive values [46]. We used the ONNX Runtime's dynamic quantization function (similar to Section 4.3), and the quantization code was as follows (note: not included in the GitHub repository):

```
model_fp32 = 'secret_model.onnx' % Path to FP32
    onnx-model
model_quant = 'secret_model_int8.onnx' % Path to
    save quantized model
quantized_model = quantize_dynamic(model_fp32,
    model_quant, weight_type=QuantType.QUInt8)
```

After quantization, we deployed the compressed model to the Samsung device and remeasured inference, see Figure 14b. As seen in the figure, the UINT8 model achieved an inference time of 48 milliseconds; a speed-up of 35 milliseconds, corresponding to a $1.7x$ speed-up. While the speed-up benefits from quantization in Section 4 showed limited potential, this experiment demonstrates that quantization can yield substantial performance gains in edge deployment. However, the model's post-quantization accuracy was not evaluated in this experiment, so any accuracy degradation remains unknown.

6. Monitoring

This lecture focused on the importance of monitoring in ML systems; both in terms of monitoring model performance, but also tracking operational metrics such as cost and environmental impact. The following subsections elaborate on each aspect.

6.1. Monitoring Model Performance

In traditional software development, a product that has been tested and deployed is generally expected to function as intended over time [1], [47]. In contrast, ML systems are inherently dynamic, as they are trained on historical data that may gradually become unrepresentative of the current real world. Consequently, model performance can degrade unless the system adapts to new patterns or distributions [48], [49]. This degradation is primarily driven by to reasons: Concept Drift and Data Drift.

Concept drift refers to a change in the relationship between the input data and the target variable over time [50], [47]. During training, ML methods learn decision boundaries from training data, assuming that the relationships between inputs and outputs remain stable. When this assumption is violated, the model's learned decision boundaries no longer apply to the current environment. Formally, it is defined as a change in $P(Y|X)$ but with no change in $P(X)$, where $P(Y|X)$ denotes the distribution of the targets Y given the inputs X. For instance, a spam classifier trained on emails from 2018 may perform poorly in 2023 since the words used by spammers may have changed over time. **Data drift**, on the other hand, involves a shift in the distribution of the inputs themselves, while the relationship between inputs and the target remains constant. In this case, $P(X)$ is changed, while $P(Y|X)$ remains the same. For example, an image-based car model classifier trained on 2020 versions of a car may struggle with newer versions due to redesigns (i.e., the input has changed but the target is still the same car model).

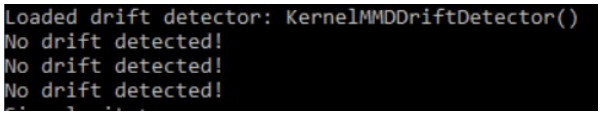
Both types of drift can negatively affect model performance [50], [51]. In the case of data drift, performance degradation is not guaranteed, as robust models may still generalize well. However, in worst case scenario, retraining or recalibration is often necessary to restore performance and make the model adapt to a new data distribution. In contrast, concept drift almost always requires intervention, including retraining with newly labeled data, or potentially developing an entirely new model architecture better suited to the new data.

6.1.1 Detecting Drift

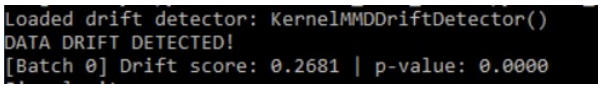
As discussed, implementing ML monitoring is essential to mitigate performance degradation over time. For our proof-of-concept, we focused exclusively on detecting data drift. To this end, we adopted the *Torchdrift* library, as it allowed us to seamlessly integrate data drift detection into our existing PyTorch scripts [52]. Torchdrift provides several predefined data drift detectors that can be calibrated on a reference dataset (typically the training data) and used to monitor for drift during inference. These detectors return a statistical p-value between 0 and 1, where lower values indicate higher likelihood of drift. Specifically, we utilized the *KernelMMDDriftDetector*, which is an image-based drift detector. The detector was calibrated using features extracted from our training data via the backbone ResNet50 model (i.e., feature extraction layers), allowing the detector to learn the training data's distribution. This calibration process is depicted in Figure 15, and the implementation can be found in the script *drift_detection/drift_detection_calibration.py* (lines 71-79) in the GitHub repository.

Figure 15. Screenshot of the terminal output from fitting (calibrating) the KernelMMD drift detector on the training data.

After fitting the detector, it was saved as an artifact, enabling reuse in a different environment. We then created an inference script (see *drift detection/inference with drift.py*) and loaded the calibrated drift detector to perform drift checks during inference. To simulate data drift, Gaussian blur was applied to the images, following Torchdrift's official example [53]. A p-value threshold of 0.01 was defined: if the detector returned a value below this threhold, data drift was flagged. Figure 16 shows terminal output when drift is absent and when it is detected.



(a) Output when no drift has been detected.



(b) Output when drift is detected.

Figure 16. Terminal outputs illustrating the absence (top) and presence of drift (bottom).

Nonetheless, a significant limitation emerged during experimentation. The training set from the Stanford car Dataset included numerous car models with varying lighting conditions and backgrounds, leading to the drift detector consistently flagging images as drifted, even when evaluated on the same dataset it was fitted on. This behavior revealed a limitation of the KernelMMD-based approach for car classification tasks. To mitigate this issue in our experiment, we restricted the calibration and evaluation to images of a single car model. While this limitation reduces the generalization capabilities and use case of the detector for real-world applications, the goal here was to demonstrate the concept of drift detection rather than provide a fully functional drift detector.

Although integrating ML monitoring through frameworks such as Prometheus or Grafana would have strengthened observability in a production setting, this feature was not implemented in the current proof-of-concept due to unresolved version compatibility issues. Initial attempts were made to resolve these conflicts by selecting non-conflicting versions aligned with the existing package dependencies. However, this approach introduced additional incompatibilities further down the pipeline. As a result, the integration was set aside. Given that the primary objective was to demonstrate the feasibility of drift detection in a controlled setup, the simplified design was considered sufficient. In a real-world deployment scenario, however, resolving these conflicts would be essential to enable robust and continuous monitoring and alerting.

In a real-world deployment scenario, if data drift is detected, I would follow the mitigation steps outlined in [51] to minimize the risk of performance degradation. The first step would involve analyzing the changes in data distribution to identify possible causes of the drift, while simultaneously monitoring model performance metrics (e.g., error rate) over time. If performance is found to degrade significantly (e.g., a drop exceeding a 10% relative to pre-drift levels), this would trigger a retraining of the model using the new data. However, as discussed in Section 2, our current setup does not support automated retraining. Assuming it was possible, I would opt for Continual Learning methods (further explored in Section 8), to incrementally update the model without removing previously acquired knowledge that may still be of relevance. If continual learning fails to restore performance, rebuilding the model would be the next step; potentially by adopting a different architecture or revisiting the feature engineering process. As a final fallback, I would consider temporarily make manual decisions without ML, or simply excluding the model in situations where it yields low performance.

6.2. Monitoring Carbon Footprint

Once ML systems are deployed, whether in the cloud, on edge devices, or on self-hosted on-premise servers (recall Section 5), they begin to consume electricity [54], [55]. This electricity may be sourced from both renewable and

non-renewable energy sources. Non-renewable energy sources (e.g., coal and petrol) do not replenish naturally and emit CO_2 along with other greenhouse gases such as methane and nitrous oxide, contributing to global warming. While renewable energy sources typically result in lower emissions, they are not entirely carbon-free. For example, the combustion of biomass (e.g., plants and algae) still releases greenhouse gases. As such, all ML systems leave a carbon footprint, which is influenced by various factors including the model's FLOP-count, dataset size, time of day, and weather conditions [47]. This highlights the motivation for investigating the carbon footprint of our model.

To track the carbon footprint of our ML model, we employed *carbontracker*, a tool developed by Anthony et al. [56] for tracking and predicting the energy consumption and carbon emissions of ML systems. The tool is publicly available on GitHub [57]. Our first measurement targeted the training sessions. Figure 17 presents the carbon footprint of the *train.py*-script. Note that this measurement was conducted using the non-scaled training script and a different setup. Consequently, the recorded training time is longer than that observed in Section 3.

```
CarbonTracker:
Actual consumption for 3 epoch(s):
        Time:   0:09:02
        Energy: 0.011645575724 kWh
        CO2eq:  1.766055052211 g
        This is equivalent to:
        0.016428419090 km travelled by car
CarbonTracker: Finished monitoring.
```

Figure 17. Carbon footprint of *train.py*.

As seen in the figure, approximately nine minutes of training resulted in an estimated energy consumption of *0.012* kWh, corresponding to a CO_2 equivalent (eq) of 1.77 grams, which is comparable to the emissions from a car traveling 0.016 km (= 16 m). Given the relatively small dataset (only 8144 training images) and the lightweight model architecture, in contrast to modern transformers, the overall carbon footprint remains modest.

We also measured the carbon footprint during inference, as computed by the *inference.py*-script. Figure 18 presents the per-sample carbon footprint during inference.
As observed in Figure 18, inference per sample consumed 3.66×10^{-6} kWh and emitted around 5.56×10^{-4} grams of CO_2, indicating a low per-inference carbon footprint.

To contextualize this, a hypothetical deployment scenario is considered, where the car classification model is used in a

```
CarbonTracker:
Actual consumption for 1 epoch(s):
        Time:   0:00:00
        Energy: 0.000003666965 kWh
        CO2eq:  0.000556096298 g
        This is equivalent to:
        0.000005172989 km travelled by car
CarbonTracker: Finished monitoring.
```

Figure 18. Carbon footprint of *inference.py*.

real-time surveillance application, continuously processing a video stream over one year. A car classifier could be useful for environmental monitoring or, more specifically, useful for monitoring cars in low-emission zones. Furthermore, the car classifier could help identify older, non-compliant models to trigger penalties or restrict access. Assuming a 24/7 video feed for 365 days at 30 frames per second, which is considered a typical frame rate for surveillance cameras [58], and instantaneous inference (justified by the millisecond-scale inference time in Figure 18), the model would annually process approximately $31,556,926 \times 30 = 946,707,780$ image frames. Using the per-frame carbon estimates, the yearly carbon footprint is presented in Figure 19.

Energy (kWh):
$Energy(946707780) = 3471.544294$

CO2 eq (g):
$CO2\_eq(946707780) \xrightarrow{\text{at 5 digits}} 526460.$

Distance by car (km):
$car(946707780) \xrightarrow{\text{at 5 digits}} 4897.3$

Figure 19. Estimated energy consumption, CO_2 emissions, and car-equivalent distance for running the car classifier continuously for one year.

As seen in Figure 19, one year of continuous operation would consume approximately *3471.54* kWh and emit *526,460* grams of CO_2, equivalent to driving a car roughly *4897.3* km. To put it into perspective, estimates suggest that ChatGPT consumes about 3 Wh (= 0.003 kWh) of energy and emit 2-3 grams of CO_2 **per query** [59]. Matching our model's annual energy consumption would require approximately $3,157,180$ queries, while matching its CO_2 emmissions would take about *175,467* queries. However, ChatGPT reportedly handles over a billion queries **daily** [60]. As such, while the yearly footprint of our model seems substantial, it is minor in comparison to a single day's usage of ChatGPT, confirming that our model has a small carbon footprint.

7. Guest Lecture

This lecture was a guest lecture delivered by Morten Lantow, Head of MLOps at DSV, in which he shared his experience and perspective on how MLOps is used in the industry [61]. Although much of the lecture centered around LLMs, which are not directly applicable to the this project, his insights offer valuable guidance for my future career path and projects in MLOps. Thus, the main takeways from the lecture are presented in this section.

According to Lantow, MLOps serves as a crucial bridge between data science and real-world applications [61]. Without MLOps, deploying AI solutions to solve business cases would not be feasible, and organizations would be unable to obtain a return on their AI investments. He identified two primary ways AI systems can deliver value:

1. Optimizing internal business processes.
2. To create an AI product.

In both cases, the business case ultimately depends on financial returns [61]. AI used for internal optimization can help reduce operational costs, while AI produce can generate direct revenue. Currently, AI is more commonly used for internal process optimization, as many manual or semi-automated processes remain inefficient and costly. However, carelessly deploying large models, particularly LLMs, can be expensive due to high computational demands, energy consumption, and carbon footprint.

When evaluating whether to deploy an AI system, DSV primarily considers two quality checks:

1. Does the AI model behave as expected in terms of performance?
2. Is the model sufficiently fast for deployment?

In practice, deployment decisions must align with business value [61]. Even if a model achieves high performance, it may still be unsuitable for deployment if it introduces bias, consumes excessive resources, or fails to meet latency requirements. He further added the importance of considering the throughput and latency if an AI model is to be deployed. Lantow also emphasized the importance of cost traceability to justify deploying AI models. For example, businesses must decide whether a $10,000 investment is worth a 1% increase in performance.

Lantow also highlighted the importance of collaboration across departments in the industry. Typically, data scientists develop model artifacts, while software engineers are responsible for deploying them into production. However, it is necessary that all teams write code in a standardized, maintainable way to avoid inefficiencies, redundancies and high cost [61].

In terms of AI infrastructure, Lantow introduced an AI deployment stack, which reflects increasing complexity:

- **Classic MLOps**: Involves deploying simpler models via API endpoints for inference.
- **Static GenAI Models**: Pre-trained LLMs that require a robust inference and serving infrastructure.
- **Retrieval-Augmented Generation (RAG)**: Combines LLMs with real-time retrieval from databases or knowledge graphs, ensuring that historical/external context is available at inference time, necessitating robust memory and context management.
- **Agentic Systems**: Autonomous systems capable of planning, decision-making and use RAG to to improve decision-making. These systems require full governance frameworks to ensure accountability, traceability , and controlled executions.

While agentic systems are still new to the industry, Lantow predicted they will become increasingly common due to their potential in reducing operational time of processes and costs [61].

As a final recommendation, Lantow advised MLOps practitioners to begin with pre-made MLOps tools:

> ”*Start by using tools that can do many of the things for you, rather than starting from scratch. In this way, you can at least get something ’out’ into the real world. The process or platform can always get optimized later. [...]. While these tools usually do not scale well, it is a good starting point*” [61]

He concluded by encouraging us to learn Kubernetes and to build a solid understanding of CI/CD pipelines; skills essential for scalable, production-level MLOps.

8. Post Deployment

In addition to ML-monitoring and maintenance (Section 6), deployed ML models may face challenges, like the need to adapt to changing environments. This was the focus of this lecture. Once deployed, a model may need to learn new patterns or unlearn existing knowledge [62]. These concepts are referred to as *continual learning* and *unlearning*, respectively [63], [64]. Each presents unique challenges that are discussed below. While the lecture exercises were not directly tied to this project, the underlying concepts are broadly relevant to any ML application.

8.1. Continual Learning

Continual learning, also known as sequential learning or lifelong learning, involves making an ML-model acquire new knowledge over time on the combined old and new datasets [63], [62]. As noted in Section 3, training large models from scratch is computationally intensive and can require significant time to train. Continual learning addresses this by enabling ML models to learn new tasks sequentially without having to retrain from scratch each time. When applying continual learning, the model is first trained on one task before being adapted to subsequent tasks. A key challenge of continual learning, however, is retaining prior knowledge while making the model acquire new information. This problem, known as *Catastrophic Forgetting*, occurs when learning on new data degrades performance on previously learned tasks. For instance, although a model may initially perform well on an earlier task, subsequent training can cause it to "forget" what it previously learned, leading to an increase in loss for earlier tasks.

To illustrate catastrophic forgetting, we used a pre-made PyTorch GitHub repository that trains and evaluates a simple Convolutional Neural Network (CNN) on the MNIST dataset to classify images of handwritten digits (0-9) [65]. To monitor performance and metrics during training, we integrated WandB into the repository. Other training hyperparameters were left unchanged; the model was trained for 14 epochs using the Negative Log Likelihood (NLL) loss. In the first phase, the CNN was trained solely on digits *zero* to *four*, and its performance was evaluated on that subset, see Figure 20. Accuracy on the remaining digits (*five* to *nine*), along with the loss for both tasks, is provided in Appendix C.1. Henceforth, I refer to classification of digits *zero* to *four* as **Task 1**, and digits *five* to *nine* as **Task 2**.

As observed in Figure 20, the model achieved an accuracy of approximately 99.8% on Task 1. To simulate catastrophic forgetting, we saved the trained weights and retrained the model on Task 2. The model's performance was then evaluated on both tasks to reveal the effects of catastrophic forgetting, see Figure 21. The model's loss curves for both tasks are available in Appendix C.1.

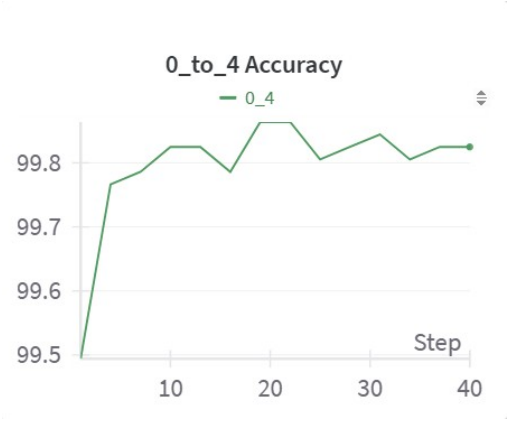
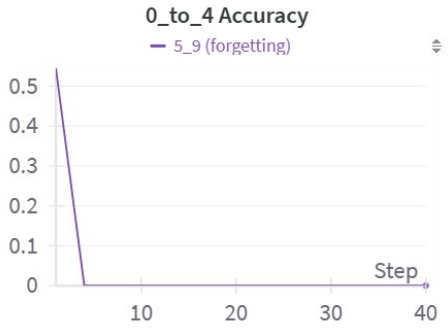
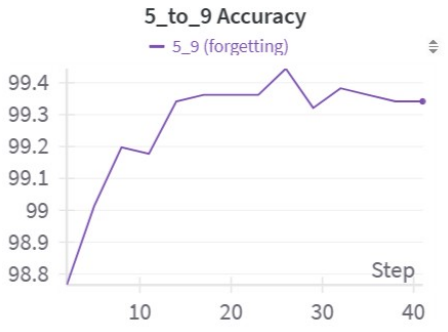


Figure 20. The CNN-model's accuracy on the first five digits when only trained on the first five digits from the MNIST dataset.



(a) The model's accuracy on the first task.



(b) The model's accuracy on the second task.

Figure 21. Plots of the model's accuracy on the first task (top) and the second task (bottom) after retraining the model using only samples from the second task. Notice how the accuracy quickly approaches 0% for the first task, demonstrating catastrophic forgetting.

As shown in Figure 21, retraining the model solely on samples from the last five digits (Task 2) caused the accuracy on the first five digits (Task 1) to quickly drop to 0%. Meanwhile, the model achieved approximately 99.3% accuracy on Task 2. This contrast clearly demonstrates catastrophic forgetting, where the model entirely lost its previously acquired knowledge from Task 1. This highlights the need for methods that enable the model to learn new tasks while mitigating the risk of catastrophic forgetting.

There are several ways to address catastrophic forgetting. Two main approaches are *Replay*-based and *Regularization*-based [63], [62]. Replay-based methods involve storing a small subset of data from previous tasks and reintroducing it during training on new tasks. This replay buffer serves as a memory mechanism, helping the model retain earlier knowledge while acquiring new information. By reintroducing past data during training on new data, the model is less likely to overwrite past knowledge. Regularization-based methods instead penalize deviations from previously learned parameters by introducing constraints to the loss function. These constraints help preserve crucial weights that are important for previous tasks. A common technique is Elastic Weight Consolidation (EWC), proposed by Kirkpatrick et al. [66], which estimates the importance of each model weight to previous tasks. During training on new tasks, changes to these weights are penalized more heavily, ensuring the preservation of knowledge that is crucial for past tasks. Less important weights remain free to adapt and change. The degree of retention is controlled by the regularization coefficient λ, which controls the balance between retaining past knowledge and adapting to new tasks.

To demonstrate the aforementioned techniques for addressing catastrophic forgetting, we implemented both Replay-based methods in conjunction with EWC. This was achieved using the Avalanche framework, which provides easily integrable plugins for Replay-based continual learning and EWC [67]. Furthermore, since Avalanche is built on top of PyTorch, it offered seamless integration with the PyTorch MNIST example, which was reemployed for this demonstration. Specifically, we utilized Avalanche's *ReplayPlugin* and *EWCPlugin*, storing 1000 samples from the first five digits and setting the λ-parameter to *0.001*. The value of λ was based on a code example provided by Avalanche. We repeated the previous sequential learning experiment; first training the CNN method on Task 1, followed by training on Task 2, but this time with Replay and EWC applied. The model's accuracy and loss for both tasks after training solely on samples from Task 1 are presented in Table 7. It is worth noting that we initially intended to use WandB to visualize the accuracy and loss across epochs. However, package versioning issues prevented successful integration of WandB into the Avalanche framework. As a workaround, terminal output screenshots of the experiment have been included in Appendix C.1 to prove exercise completion.

Performance on Task 1 Samples		
Task	**Accuracy**	**Loss**
1 (zero to four)	99.6%	0.018
2 (five to nine)	0%	16.06

Table 7. The table shows the model's performance of both tasks after completing training on the first five digits.

As shown in Table 7, the model achieved high accuracy on Task 1, while its performance on the second task remained 0%, indicating that it had not yet been trained on digits 5-9, as expected. After completing sequential training on samples from Task 2 (with Replay and EWC), the model's updated performance is shown in Table 8.

Performance on Task 2 Samples (w/Replay and EWC)		
Task	**Accuracy**	**Loss**
1 (zero to four)	97.5%	0.093
2 (five to nine)	88.3%	0.66

Table 8. The table shows the model's performance of both tasks after completing training on the last five digits. Notice how the model managed to retain knowledge from the first task whilst learning the second task.

As observed in Table 8, after training on Task 2 using Replay and EWC, the model successfully retained its knowledge of Task 1, indicated by a high accuracy of 97.5% whilst also learning the new task, achieving an accuracy of 88.3% on Task 2. In contrast to earlier sequential learning experiment (Figure 21), where the model entirely forgot Task 1, applying Replay and EWC showed much more favorable results by enabling the model to learn new tasks while retaining knowledge of the previous task. Although further tuning of the Replay buffer size or the EWC λ-parameter could potentially improve performance on Task 2, the results clearly demonstrate that these techniques allow ML models to adapt to new tasks without causing a high risk of catastrophic forgetting.

8.2. Unlearning

While the previous section focused on enabling ML models to learn new tasks, there are situations where the objective is the opposite: to make a model forget certain learned information without compromising its performance on remaining tasks [64], [62]. This concept, known as *Unlearning*, is particularly relevant when a model has been

trained on sensitive personal data or copyrighted material, and legal or ethical obligations may require that this data is removed from the model's "knowledge base". However, unlike traditional databases where records can simply be deleted, information in ML models is embedded across millions of parameters. Due to the black-box nature of most ML models, it is challenging to identify and remove the specific parameters responsible for storing unwanted information [18]. Additionally, this information is often distributed across multiple weights, further complicating the removal process. This calls for techniques that can effectively enable the model to forget specific learned knowledge.

There are three primary approaches to model unlearning [64]:

1. **Retraining the model without the sensitive data:** while this straightforward and generally effective, it is also computationally expensive and time-consuming, as discussed in Section 3 and Section 6.2.

2. **Output suppression:** this approach uses guardrails (or filters) to prevent the model from generating undesirable content. For example, the Chinese LLM, DeepSpeek, has implemented such guardrails to restrict the disclosure of information censored in China [68].

3. **Unlearning algorithms:** these techniques are designed to remove specific information from a model post-training. Although promising, they are still in early development stages and lack maturity [62].

Despite these approaches, unlearning remains an imperfect solution [64]. Output suppression merely conceals sensitive content without actually "removing" it from the model. Retraining, even on a filtered dataset, may leave latent traces of the removed data. Moreover, the effectiveness of unlearning techniques depends on the generality of the concept being removed [62]. Desai et al. [69] introduced MERU, a model that generates hyperbolic representations of images and text. Their findings suggest that general concepts (e.g., "animal") tend to reside near the minimum of the hyperboloid, while more specific concepts (e.g., "cat") are positioned higher in the hierarchy. Since specific concepts occupy more localized regions in the latent space, they are relatively easier to remove from the model's knowledge base [62]. In contrast, general concepts are more deeply entangled in the latent space, thus much harder to remove.

As discussed, there is no "one-size fits all" approach to unlearning. However, a simple yet effective technique is *Gradient Ascent unlearning* [62]. DL methods acquire knowledge through backpropagation and optimization algorithms, typically based on Gradient Descent [18]. By reversing the gradient (i.e., multiplying it by -1), the model is trained to increase its error on a specific class, thereby reducing its confidence in recognizing that class and thus "unlearning" it. To demonstrate this, we applied Gradient Ascent unlearning to the MNIST dataset using the same experimental setup as described in Section 8.1. The CNN-method was initially trained on the complete MNIST dataset (i.e., all digits). We then evaluated its accuracy across all classes to establish a baseline for assessing the potential side effects of unlearning. The accuracy for each digit is presented in Table 9. WandB was used to log the accuracy of through epochs during training, see Appendix Section C.2, Figure 32. After training, we saved the model weights.

Model Performance (All Digits)	
Class (Digit)	**Accuracy**
0	99.49%
1	99.91%
2	99.32%
3	99.50%
4	99.19%
5	99.10%
6	98.96%
7	98.83%
8	98.87%
9	98.22%

Table 9. Model performance after being trained on samples of every digit in the MNIST dataset.

Next, we applied Gradient Ascent unlearning on the digit *seven*. Rather than retraining the model from scratch (an expensive process), we divided the dataset into "forget" and "retain" subsets. Samples of the digit *seven* were marked as "forget" and subjected to Gradient Ascent. The remaining digits formed the "retain" subset and were trained with standard Gradient Descent. This distinction was crucial to prevent the model from completely forgetting all knowledge. In our initial experiment, we applied Gradient Ascent only to the forget set, without simultaneously applying Gradient Descent to the retain set. As a result, the model's accuracy on all classes approached 0%. Reflecting back on the initial experiment, this behavior aligned with catastrophic forgetting, where learning a new task (here, unlearning the digit *seven*) erases prior knowledge (recall Section 8.1). Recognizing this mistake, we revised our approach: while applying Gradient Ascent to digit *seven*, we simultaneously trained on the retain set using Gradient Descent. The results are shown in Table 10, with class *seven* being marked in **bold**. As before, accuracy plots are available in Appendix C.2, Figure 33.

As observed in Table 10, the accuracy for digit *seven*

Model Performance (All Digits)	
Class (Digit)	**Accuracy**
0	99.49%
1	99.91%
2	99.81%
3	99.60%
4	99.19%
5	98.99%
6	98.94%
7	**1.07%**
8	99.18%
9	98.81%

Table 10. Model performance after unlearning the digit *seven*.

dropped from 98.83% (baseline) to 1.07%, effectively rendering the model incapable of correctly classifying it. However, since the model was still exposed to digit *seven*, the accuracy did not fall to 0%. Instead, the model became highly uncertain when encountering samples of that digit. Thus, Gradient Ascent unlearning did not entirely erase information but rather induced uncertainty about the "forgotten" class. However, the accuracy for the remaining digits remained high, showing accuracies above 98%. Comparing the baseline (Table 9) with the unlearning results (Table 10), we observe that the remaining classes maintained high accuracy. In some cases, they even showed slight improvements, possibly due to subtle shifts in feature representation following the unlearning process.

9. Conclusion

This report presents an MLOps project centered around a classifier, integrating a Continuous Integration / Continuous Deployment pipeline via GitHub Actions. Challenges emerged when attempting to automate training using AILAB. Nonetheless, the car classifier was made scalable; training was optimized using Distributed Data Parallelism, and inference efficiency was improved through quantization. Machine Learning deployment was further explored using a Samsung Galaxy S21, albeit not with the car classifier. To support monitoring, data drift was detected using the Torchdrift framework, and the model's carbon footprint was estimated, revealing a significantly lower environmental impact compared to Large Language Models, such as ChatGPT. Additionally, a guest lecture by Morten Lantow provided valuable insights into industry practices regarding Machine Learning, which can inform my future career path and projects. Continual learning was also examined, highlighting the effects of catastrophic forgetting post-deployment. Replay-based and Regularization-based strategies were applied to mitigate this challenge, allowing the model to learn new tasks while preserving knowledge of previous ones. Lastly, unlearning was demonstrated using Gradient Ascent. Although this method reduced the model's confidence in a specific class, it did not fully erase the learned information, highlighting the need for a more effective unlearning technique.

In conclusion, this report documents the completion of the MLOps course. Despite making mistakes and encountering setbacks throughout the project, these challenges offered valuable learning opportunities that will guide future work.

References

[1] Emmanuel Raj. *Engineering MLOps*. Packt Publishing, 2021. 1, 12

[2] Jarek Kazmierczak, Khalid Salama, Valentin Huerta, and Sunil Kumar Jang Bahadur. Mlops: Continuous delivery and automation pipelines in machine learning. Google Cloud, 2024. https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning. [Accessed Date: 05/02/2024]. 1, 3

[3] GeeksForGeeks. Waterfall model – software engineering, 2024. https://www.geeksforgeeks.org/waterfall-model/. [Accessed Date: 08/02/2024]. 1

[4] PyTorch. resnet50, 2017. https://pytorch.org/vision/main/models/generated/torchvision.models.resnet50.html. [Accessed Date: 08/02/2024]. 2

[5] Jesús Utrera. Stanford car dataset by classes folder. Kaggle, 2018. https://www.kaggle.com/datasets/jutrera/stanford-car-dataset-by-classes-folder. [Accessed: 12/04/2025]. 2

[6] DVC. Get started with dvc, n.d. https://dvc.org/doc/start#get-started-with-dvc. [Accessed: 06/06/2025]. 2

[7] GitHub. About protected branches, n.d. https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-protected-branches/about-protected-branches. [Accessed 13/02/2025]. 3

[8] Goku Mohandas. Pre-commit - made with ml, 2023. https://madewithml.com/. [Accessed: 13/02/2025]. 3

[9] pre commit. pre-commit: A framework for managing and maintaining multi-language pre-commit hooks., n.d. https://pre-commit.com/. [Accessed: 13/02/2025]. 3

[10] Andreas Aakerberg. Mlops lecture 2: Continuous ml, 2025. [Date of Lecture: 12/02/2025]. 3

[11] Goku Mohandas. Code - made with ml, 2023. https://madewithml.com/. [Accessed: 13/02/2025]. 3

[12] Coverage.py. Coverage.py: Documentation, 2025. https://coverage.readthedocs.io/en/7.6.12/. [Accessed: 15/02/2025]. 3

[13] GitHub. Github actions: Automate your workflow from idea to production, n.d. https://github.com/features/actions. [Accessed: 13/02/2025]. 4

[14] CLAAUDIA. Ci/cd with github actions. AAU HPC, n.d. https://hpc.aau.dk/ai-lab/guides/ci-cd-with-github-actions/. [Accessed: 21/04/2025]. 4

[15] Andreas Aakerberg. Mlops lecture 3: Scalable training, 2025. [Date of Lecture: 19/02/2025]. 5, 6, 7, 8

[16] Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. arXiv, 2020. 5

[17] Fiveable Inc. Amdahl's law and gustafson's law, 2024. https://library.fiveable.me/parallel-and-distributed-computing/unit-8/amdahls-law-gustafsons-law/study-guide/5w3ckhKQ6tq5bfql. [Accessed: 01/03/2025]. 5, 6

[18] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. https://D2L.ai. 6, 18

[19] Maxim Lukiyanov, Guoliang Hua, Geeta Chauhan, and Gisle Dankel. Introducing pytorch profiler – the new and improved performance tool. PyTorch, 2021. https://pytorch.org/blog/introducing-pytorch-profiler-the-new-and-improved-performance-tool/ [Accessed: 12/05/2025]. 6

[20] Mirza Mujtaba. Distributed training: Guide for data scientists. Neptune.ai, 2023. https://neptune.ai/blog/distributed-training. [Accessed: 02/03/2025]. 6

[21] PyTorch. torchrun (elastic launch), 2024. https://pytorch.org/docs/stable/elastic/run.html. [Accessed: 02/03/2025]. 7

[22] CalculatorSoup. Percentage difference calculator, 2025. https://www.calculatorsoup.com/calculators/algebra/percent-difference-calculator.php. [Accessed: 17/05/2025]. 7

[23] Enes Zvornicanin. Relation between learning rate and batch size. Baeldung, 2025. https://www.baeldung.com/cs/learning-rate-batch-size . [Accessed: 22/04/2025]. 7

[24] GeeksForGeeks. How should the learning rate change as the batch size changes?, 2024. https://www.geeksforgeeks.org/how-should-the-learning-rate-change-as-the-batch-size-changes/. [Accessed: 22/04/2025]. 7

[25] NVIDIA. Automatic mixed precision for deep learning, 2025. https://developer.nvidia.com/automatic-mixed-precision. [Accessed: 02/03/2025]. 8

[26] Rangan Majumder and Junhua Wang. Zero deepspeed: New system optimizations enable training models with over 100 billion parameters. Microsoft, 2020. https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/. [Accessed: 02/03/2025]. 8, 9

[27] PyTorch. Automatic mixed precision package - torch.amp, n.d. https://docs.pytorch.org/docs/stable/amp.html. [Accessed: 06/06/2025]. 8

[28] DeepSpeed. Zero redundancy optimizer, 2025. https://www.deepspeed.ai/tutorials/zero/. [Accessed: 06/06/2025]. 8

[29] Song Bian, Minghao Yan, and Shivaram Venkataraman. Scaling inference-efficient language models. ArXiv, 2025. 9

[30] Andreas Aakerberg. Mlops lecture 4: Scalable inference, 2025. [Date of Lecture: 05/03/2025]. 9, 10

[31] Alexander S. Gillis. Latency. TechTarget, 2025. https://www.techtarget.com/whatis/definition/latency. [Accessed: 09/03/2025]. 9

[32] John Burke. What is throughput? TechTarget, 2025. https://www.techtarget.com/searchnetworking/definition/throughput [Accessed: 09/03/2025]. 9

[33] Apple Inc. Adjusting for gpu memory bandwidth tradeoffs, 2025. https://developer.apple.com/documentation/metal/adjusting-for-gpu-memory-bandwidth-tradeoffs. [Accessed: 09/03/2025]. 9

[34] Neural Network Distiller. Quantization, n.d. https://intellabs.github.io/distiller/quantization.html. [Accessed: 09/03/2025]. 9, 10

[35] Neural Network Distiller. Pruning, n.d. https://intellabs.github.io/distiller/pruning.html. [Accessed: 09/03/2025]. 10

[36] Neural Network Distiller. Knowledge distillation, n.d. https://intellabs.github.io/distiller/knowledge_distillation.html [Accessed: 09/03/2025]. 10

[37] PyTorch. Quantization, 2025. https://docs.pytorch.org/docs/stable/quantization.html. [Accessed: 06/06/2025]. 10

[38] HDCharles. (prototype) gpu quantization with torchao, 2024. https://docs.pytorch.org/tutorials/prototype/gpu_quantization_torchao_tutorial.html?highlight=transformer. [Accessed: 06/06/2025]. 10

[39] Hopsworks.ai. Batch inference pipeline, 2025. https://www.hopsworks.ai/dictionary/batch-inference-pipeline. [Accessed: 09/03/2025]. 10

[40] FastAPI. Fastapi documentation, n.d. https://fastapi.tiangolo.com/#license. [Accessed: 06/06/2025]. 10

[41] NVIDIA. Nvidia triton inference server, 2025. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html. [Accessed: 06/06/2025]. 11

[42] Tiffany Yeung. What's the difference between edge computing and cloud computing? NVIDIA, 2022. https://blogs.nvidia.com/blog/difference-between-cloud-and-edge-computing/. [Accessed: 15/03/2025]. 11

[43] Muhammad Asim, Yong Wang, Kezhi Wang, and Pei-Qiu Huang. A review on computational intelligence techniques in cloud and edge computing. IEEE Xplore, 2020. 11

[44] Andreas Aakerberg. Mlops lecture 5: Deployment, 2025. [Date of Lecture: 11/03/2025]. 11

[45] Microsoft. Github repository: onnxruntime-inference-examples. GitHub, n.d. https://github.com/microsoft/onnxruntime-inference-examples/tree/main/mobile/examples/image_classification/android. [Accessed: 15/03/2025]. 11

[46] Embedded Wizard. Data types: uint8, uint16, uint32, uint64, 2025. https://doc.embedded-wizard.de/uint-type. [Accessed: 15/03/2025]. 12

[47] Andreas Aakerberg. Mlops lecture 6: Monitoring, 2025. [Date of Lecture: 19/03/2025]. 12, 14

[48] Elena Samuylova. Machine learning monitoring, part 1: What it is and how it differs. Evidently, 2025. https://www.evidentlyai.com/blog/machine-learning-monitoring-what-it-is-and-how-it-differs. [Accessed: 26/03/2025]. 12

[49] Emeli Dral and Elena Samuylova. Machine learning monitoring, part 5: Why you should care about data and concept drift. Evidently, 2025. https://www.evidentlyai.com/blog/machine-learning-monitoring-data-and-concept-drift. [Accessed: 26/03/2025]. 12

[50] Evidently AI Team. What is concept drift in ml, and how to detect and address it. Evidently, 2025. https://www.evidentlyai.com/ml-in-production/concept-drift. [Accessed: 29/03/2025]. 12

[51] Emeli Dral and Elena Samuylova. My data drifted. what's next?" how to handle ml model drift in production. Evidently, 2025. https://www.evidentlyai.com/blog/ml-monitoring-data-drift-how-to-handle. [Accessed: 29/03/2025]. 12, 13

[52] Thomas Viehmann, Luca Antiga, Daniele Cortinovis, and Lisa Lozza. Torchdrift: drift detection for pytorch, 2021. https://torchdrift.org/index.html. [Accessed: 05/06/2025]. 13

[53] Thomas Viehmann, Luca Antiga, Daniele Cortinovis, and Lisa Lozza. Drift detection on image classifiers, 2021. https://torchdrift.org/notebooks/drift_detection_on_images.html. [Accessed: 05/06/2025]. 13

[54] Kasper Groes and Albin Ludvigsen. How to estimate and reduce the carbon footprint of machine learning models. Towards Data Science, 2022. https://towardsdatascience.com/how-to-estimate-and-reduce-the-carbon-footprint-of-machine-learning-models-49f24510880/. [Accessed: 29/03/2025]. 13

[55] Victor Schmidt, Alexandra Luccioni, Alexandre Lacoste, and Thomas Dandres. Ml co2 impact. ML CO2 Impact, n.d. https://mlco2.github.io/impact/. [Accessed: 29/03/2025]. 13

[56] Lasse F. Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. ICML Workshop on Challenges in Deploying and monitoring Machine Learning Systems, July 2020. 14

[57] Ifwa. carbontracker. GitHub, 2020. https://github.com/lfwa/carbontracker. [Accessed: 29/03/2025]. 14

[58] ProtectFind. Security camera frame rate (fps) explained, 2025. https://protectfind.com.au/security-cameras/frame-rate-fps/. [Accessed: 11/05/2025]. 14

[59] Hannah Ritchie. What's the carbon footprint of using chatgpt? Sustainability By Numbers, 2025. https://www.sustainabilitybynumbers.com/p/carbon-footprint-chatgpt. [Accessed: 11/05/2025]. 14

[60] NerdyNav. 107+ chatgpt statistics and user numbers, 2025. https://nerdynav.com/chatgpt-statistics/. [Accessed: 11/05/2025]. 14

[61] Morten Lantow. Guest lecture by morten lantow, 2025. [Date of Lecture: 26/03/2025]. 15

[62] Andreas Aakerberg. Mlops lecture 8: Post deployment, 2025. [Date of Lecture: 02/04/2025]. 16, 17, 18

[63] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. ArXiv, 2015. 16, 17

[64] A. Feder Cooper et al. Machine unlearning doesn't do what you think: Lessons for generative ai policy, research, and practice. ArXiv, 2024. 16, 17, 18

[65] PyTorch. Distributeddataparallel, 2024. https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html. [Accessed: 02/03/2025]. 16

[66] James Kirkpatrick, Razvan Pascanua, Neil Rabinowitza, Joel Venessa, Guillaume Desjardinsa, Andrei A. Rusua, Kieran Milana, John Quana, Tiago Ramalhoa, Agnieszka Grabska-Barwinska, Demis Hassabisa, Claudia Clopathb, Dharshan Kumarana, and Raia Hadsella. Overcoming catastrophic forgetting in neural networks. ResearchGate, 2017. 17

[67] Vincenzo Lomonaco, Lorenzo Pellegrini, Andrea Cossu, Antonio Carta, Gabriele Graffieti, Tyler L. Hayes, Matthias De

Lange, Marc Masana, Jary Pomponi, Gido van de Ven, Martin Mundt, Qi She, Keiland Cooper, Jeremy Forest, Eden Belouadah, Simone Calderara, German I. Parisi, Fabio Cuzzolin, Andreas Tolias, Simone Scardapane, Luca Antiga, Subutai Amhad, Adrian Popescu, Christopher Kanan, Joost van de Weijer, Tinne Tuytelaars, Davide Bacciu, and Davide Maltoni. Avalanche: an end-to-end library for continual learning. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2nd Continual Learning in Computer Vision Workshop, 2021. 17

[68] Mehul Gupta. Deepseek is highly biased, don't use it. Medium, 2025. https://medium.com/data-science-in-your-pocket/deepseek-is-highly-biased-dont-use-it-2cb0358647f9 [Accessed: 08/04/2025]. 18

[69] Karan Desai et al. Hyperbolic image-text representations. ArXiv, 2024. 18

A. Scalable Training: Additional Details

This section provides additional details for the following from Section 3:

1. Intra-Node Experiments
2. ZeRO Experiment

A.1. Intra-Node Experiments

In this section, screenshots of GPU utilization, when measuring the training time with different batch sizes, are presented. Note, the screenshots are snapshots of the GPU utilization; in practice it fluctuates.

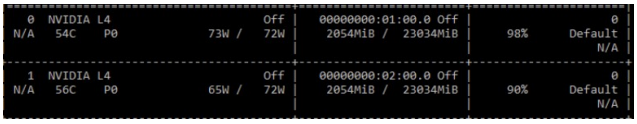


Figure 22. GPU utilization with a batch size of 8

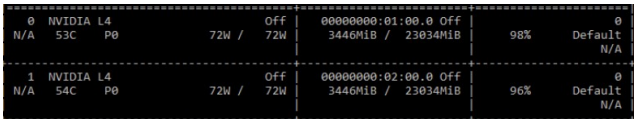


Figure 23. GPU utilization with a batch size of 18.

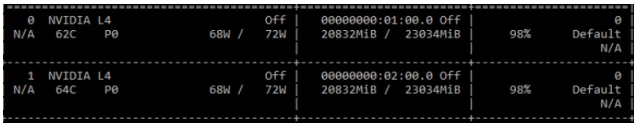


Figure 24. GPU utilization with a batch size of 20.

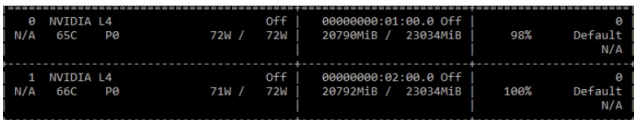


Figure 25. GPU utilization with a batch size of 26.

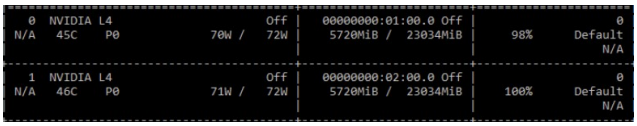
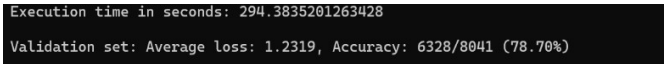


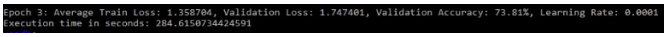
Figure 26. GPU utilization with a batch size of 32.

A.2. Accuracy Across Different Batch Sizes

The validation accuracy score for the configuration using a batch size of 16 and 32 are presented in Figure 27.



(a) Batch size 16 accuracy



(b) Batch size 32 accuracy

Figure 27. Plots of the ResNet50 model's accuracy when using a batch size 16 (top) and 32 (bottom).

A.3. ZeRO Experiment

When using ZeRO, we noticed significant slower training times with batch sizes of 16 and 18. Therefore, we investigated the GPU utilization of batch size 18 during Stage 1, which is presented in Figure 28.

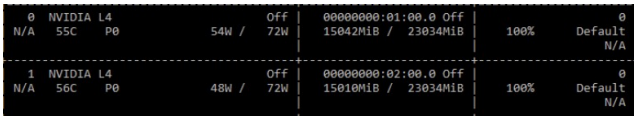


Figure 28. GPU utilization of Stage 1

As observed in Figure 28, the GPU utilization for both GPUs was 100%, indicating the slow training times likely stem from communication overhead, especially in the later stages.

Furthermore, since we were forced to halve the precision to FP16, the model experienced a slight drop in accuracy, which can be seen in Figure 29. The figure illustrates the performance of the batch size 16-configuration in Stage 3.

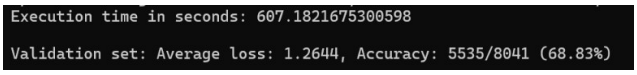


Figure 29. Accuracy of the model with a batch size of 16 in Stage 3 of ZeRO

B. Scalable Inference: Additional Details

The measurements used to create Figure 13 in Section 4.3 are presented in Table 11.

Batch size	Latency per Batch (ms)	Throughput (images/second)
1	83.89	11.92
2	170.15	11.75
4	364.18	10.98
8	759.81	10.53
16	1492.79	10.34
32	2992.73	10.31

Table 11. Table showcasing the latency and throughput for each batch during batch inference.

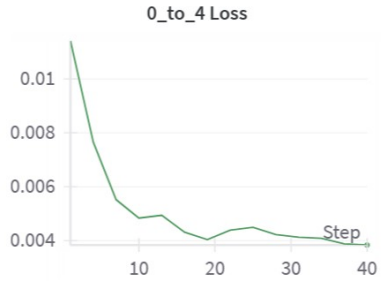
C. Post Deployment: Additional Details

This section provides additional details on the following from Section 8:

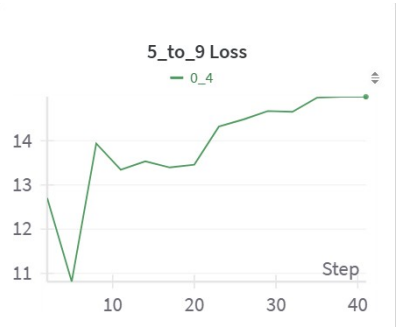
1. Continual learning (Section 8.1)
2. Unlearning (Section 8.2)

C.1. Continual Learning: Additional Details

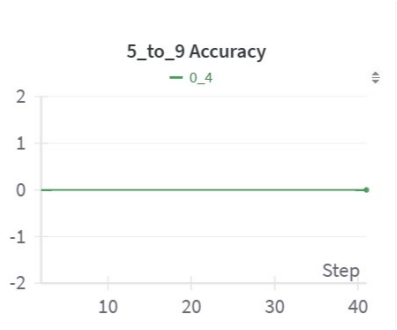
The model's loss and accuracy (for both tasks) during training on samples from the first five digits can be seen in Figure 30.



(a) The model's NLL loss on the first five digits (zero to four)



(b) The model's NLL loss on the last five digits (five to nine)

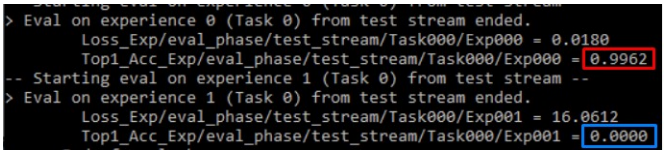


(c) The model's accuracy on the last five digits. Notice how the accuracy is 0%.

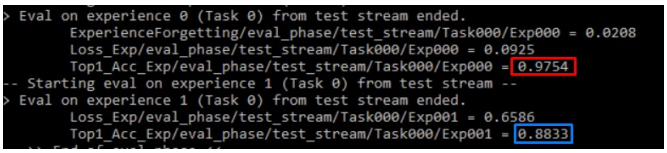
Figure 30. Additional plots for the demonstration of catastrophic forgetting

As observed in Figure 30, the NLL loss on the first five digits was approximately 0.004, whereas it was around 15 for the last five digits. This was expected, as it was not trained on samples of the last five digits. Consequentially, the accuracy for the last five digits was 0%.

We then used Replay and EWC to mitigate the risk of catastrophic forgetting. The model's performance after completing training on both the first five digits and last five digits can be seen in Figure 31. The accuracy for the first task is marked in red, whereas the accuracy for the second task is marked in blue.



(a) The model's performance after completing training on samples from the first task.



(b) The model's performance after completing training on samples from the second task.

Figure 31. The model's performance after completing training on the first task (top) and the second task (bottom) with Replay and EWC.

C.2. Unlearning: Additional Details

When establishing a baseline accuracy on all classes of the MNIST dataset, WandB was used to track the accuracy during training. The model was evaluated on a test set for every epoch, see Figure 32

As seen in Figure 32, the accuracy quickly rises above 98%. We then applied Gradient Ascent on class 7 while using standard Gradient Descent on the other classes to demonstrate unlearning. The accuracy plots can be seen in Figure 33

It can be observed that the accuracy for class 7 drops significantly from $\sim 4.1\%$ to $\sim 1\%$ during retraining with Gradient Ascent, demonstrating unlearning for class 7.

Figure 32. Model performance of each digit during training. The model was evaluated on a test set for every epoch.

Figure 33. Model performance of each digit after applying Gradient Ascent to class 7.

D. Use of Generative AI

In accordance to the rules of the Dept. of Architecture, Design, and Media Technology, it must be clearly stated if AI tools were used for Course Modules if permitted on the respective course's Moodle page, which it was for this course. Furthermore, students must describe which tools were used and how they were used (including prompts).

Throughout this project, I have used ChatGPT for grammatical revision before handing in the assignment, as well as a code-debugging tool. Below is a written prompt-example of the input it received and the output it returned.

Input:
Is this clearly explained in an academic tone?

> *"As observed in Figure X, the NLL loss on the first five digits is approximately 0.004, whereas it is around 15 for the last five digits, which is expected, as it was not trained on samples of the last five digits."*

Output:
Your sentence is mostly clear and written in an academic tone, but it could benefit from a few adjustments for precision and formality. Here's a revised version:

> *"As shown in Figure X, the negative log-likelihood (NLL) loss for the first five digits is approximately 0.004, whereas for the last five digits it is around 15. This disparity is expected, as the model was not trained on samples from the latter group."*

Rationale for changes:

- Expanded "NLL" to "negative log-likelihood (NLL)" on first mention for clarity.
- Replaced "as observed" with "as shown" — both are acceptable, but "shown" is more common in figure references.
- "Whereas for the last five digits it is around 15" improves readability by avoiding a comma splice.
- Slightly more formal tone with "latter group" and "disparity."

Using ChatGPT has helped me improving my formulations to ensure better clarity and structure. However, I wish to clarify that I have **never** fully copy/pasted ChatGPT's output; only words or single phrases were adjusted per ChatGPT's recommendations if I deemed them to improve formulation and clarity. Furthermore, the original content was written using my own words, and ChatGPT was never used to generate content. I only applied ChatGPT's changes if I would be able to write them myself. For instance, if ChatGPT used a word that I would never use in my personal writing style, I would use my own formulation instead.