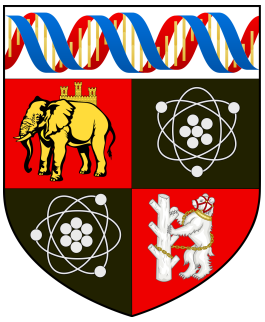
Introducing A Unified Framework for Continual Learning and Machine Unlearning

Guney Coban, Thomas Marsh, Krishi Mistry, Gerald Ramos

Supervised by Paris Giampouras and Peter Triantafillou

2024-25



Abstract

In the modern era of data privacy and evolving machine learning applications, the ability to selectively forget previously learned information has become just as important as learning new knowledge. In this report, we investigate the interplay between continual learning and machine unlearning, two areas traditionally treated as separate. We introduce NegGEM, a novel framework that demonstrates how information retained during learning can be effectively leveraged to enhance unlearning. Our work develops multiple variations of this unified approach, including Unlearning-AGEM, Random Labelling GEM/AGEM, GEM+, and ALT-NegGEM, exploring trade-offs between stability, plasticity, forgetting quality, and computational efficiency. Extensive evaluations, including Membership Inference Attack (MIA) testing, confirm that integrating learning and unlearning objectives can improve unlearning success while preserving model utility. Our findings suggest a promising direction for building more adaptable and privacy-preserving machine learning systems.

Keywords: *Continual Learning, Machine Unlearning, Neural Networks, Model Privacy, Algorithm Development, Artificial Intelligence*

Acknowledgements

We would like to thank our supervisors, Peter Triantafillou and Paris Giampouras, for their support and guidance throughout this project. Their expertise and feedback were invaluable in helping us shape and deliver our work.

We also thank the Department of Computer Science at the University of Warwick for providing the resources needed to complete the project.

Finally, we are grateful to the researchers and developers in the Continual Learning and Machine Unlearning spaces whose open-source contributions made much of this project possible.

Contents

1 Project Motivation

Over the last few years, we have seen the amount of discussion around machine learning and artificial intelligence increase significantly. As adoption increases, problems with existing methods become apparent. Training a neural network based model embeds the training data within its parameters and abstracts away the effects of individual data points. In cases where we want to tweak the dataset that our model is representing, the only way we can be sure is by retraining the model from scratch. This retraining process is expensive at the small scale, and even more so with larger data sets and models. Two scenarios where this arise are trying to build on an existing model after the initial training (Continual Learning), and removing the effect of certain data on our model (Machine Unlearning). A system that allows us to do these things would be invaluable.

Continual Learning is the idea of feeding additional data to a previously trained model so that it can extend its knowledge beyond the initial training set. A typical workflow for training a model will have the entire dataset before starting training, allowing us to make decisions about the optimal training strategy for our model before starting. In reality, we might expect to see our potential training dataset grow. Naïvely adding this to our model by training it on this new data causes "Catastrophic Forgetting" (1) where we see significantly reduced performance on tasks trained with our older dataset. An example might be a model trained to differentiate dogs from cats. By training it to also identify plants (a new task), we would expect to see the performance for identifying dogs and cats drop. Since our loss is only being computed for images of plants, the model is incentivised to sacrifice performance on the first two tasks for performance on the new one.

Although it may be possible to retrain a new model from scratch to identify all three without this degradation in performance, that will be computationally expensive. There are also cases where this is not possible; we may no longer have access to the original dataset. In either case, we have a strong incentive to update the original model, which means developing methods around the problem of Catastrophic Forgetting.

Machine Unlearning tackles the other side of modifying a pre-existing model. The aim of Machine Unlearning is to remove the effect of a subset of data from a

model (a forget set). There are a few different reasons why one may want to do this: under GDPR, one has the "right to be forgotten". In cases where someone has used that right, the owner of the model "shall have the obligation to erase personal data without undue delay" (2). The European Data Protection Board has deemed that in some scenarios "the information from the training dataset, including personal data, may still remain 'absorbed' in the parameters of the model" (3). A company that has trained a model on customer data would have to remove the influence of this data from the model to comply with the regulations. There is a caveat, however: naïve methods can open a model up to Membership Inference Attacks (MIAs). If a model forgets too aggressively, it may reveal information about the forget set through the absence or distortion of related outputs (4). Practically speaking, the data still remains "absorbed" in the model, just in a different form.

As mentioned, retraining from scratch without the offending data is expensive. A less computationally expensive method to modify the model to be as close as possible to a re-trained model is therefore valuable. Again, it may also be the case that you do not have access to the entire dataset used to train your original model, in which case there is an even stronger incentive to modify your existing model.

The main inspiration for this project is the process of Catastrophic Forgetting. Through Continual Learning, we are causing Machine Unlearning. A system built for both learning and unlearning should, in theory, let us achieve the same goals with greater efficiency and better results.

2 Problem Statements

While these areas are typically explored in isolation, we hypothesised that there may be useful overlap between the two. Our goal was to test this hypothesis through the following objectives:

2.1 Learning and Unlearning Interactions

The primary objective was to investigate how Continual Learning and Machine Unlearning interact when applied sequentially. We aimed to assess whether Continual Learning affects the effectiveness of unlearning, and vice versa, and whether both processes can be performed concurrently. Additionally, we examined whether

Continual Learning makes models more susceptible to unlearning, or the reverse.

2.2 A Unified Learning–Unlearning Algorithm

The second major task was to explore the feasibility of creating a joint algorithm capable of performing both learning and unlearning simultaneously. Traditional workflows treat these as distinct steps, each with their own optimisation routines and computational overhead. Our hypothesis was that a single, unified update rule could be developed to incorporate new data while forgetting specific older samples — ideally achieving performance comparable to, or better than, running the two operations separately.

We aimed to build and benchmark this unified method against baseline approaches across three main metrics: model performance, unlearning success, and computational efficiency. Model performance would be measured primarily through classification accuracy on both the retained and newly introduced data. Unlearning success would be evaluated using dedicated forgetting metrics, as well as privacy-focused assessments such as Membership Inference Attacks (MIAs), to determine whether the influence of the forgotten data had been successfully removed. Finally, computational efficiency would be judged based on training time and resource usage, allowing us to compare the unified approach to sequential learning–unlearning baselines in terms of practical viability.

3 Preliminaries

This section reviews the essential concepts required for the remainder of the paper. We begin with the structure of a feed-forward neural network: weight and bias parameters, the forward pass, and its matrix formulation—because Continual Learning and Machine Unlearning algorithms ultimately act on these components.

For a generalised high level overview of a neural network, a trained neural network consists of layers containing weights and optionally, biases which are optimised when training a neural network for tasks such as classification. In the case of the neural network in Figure 1, a simple example provides a binary classification tasks provided an input vector X containing 3 elements x_1, x_2 and x_3.

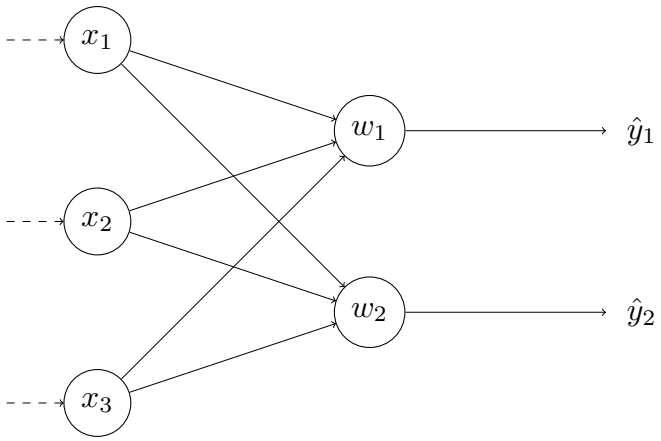


Figure 1: Simple example Feed Forward Neural Network

Therefore, in order to utilise the weight vector W to perform an instance of forward propagation i.e. performing a prediction to by generating the output vector $\hat{Y}$ we perform the following matrix operations.

$$\hat{Y} = XW^T + B \tag{3.1}$$

In this case, the addition of a bias vector B is optional and depends on if the task calls for it in conjunction with the use of non-linear activation functions such as ReLU.

3.1 Deep Neural Networks

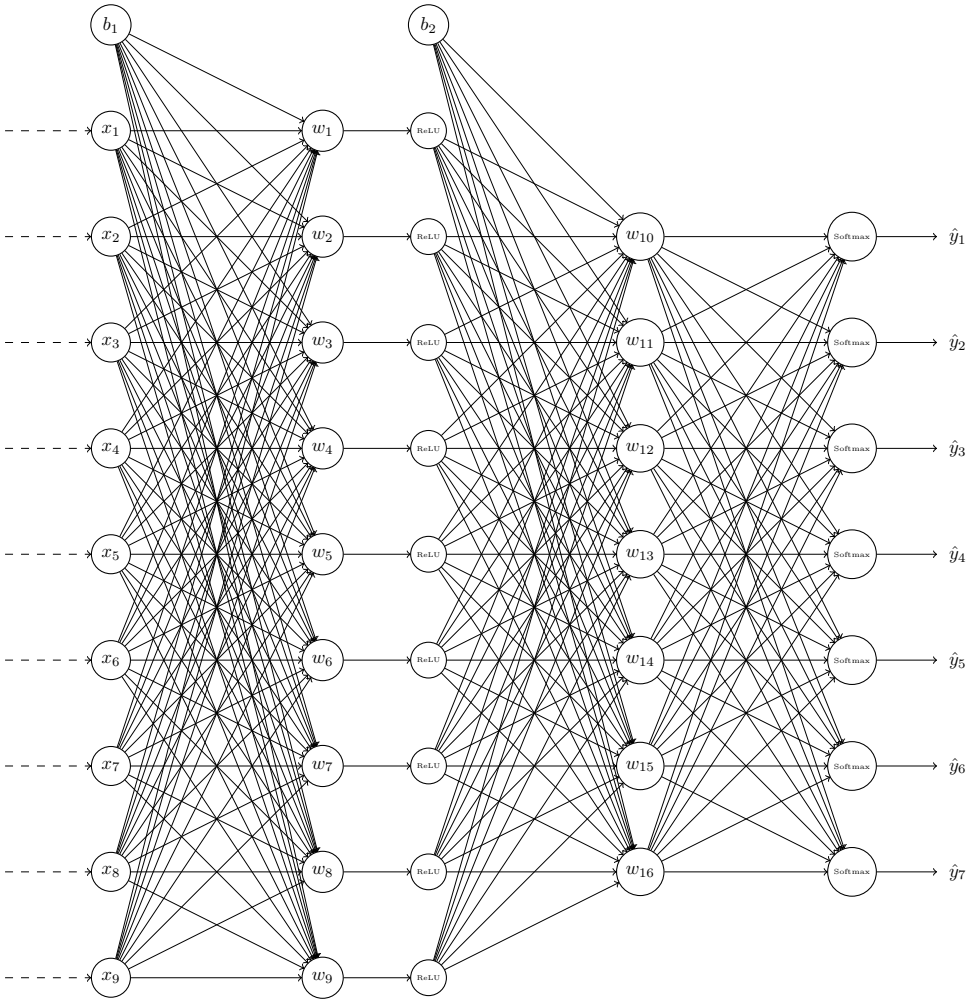


Figure 2: Example 9-Input 7-Class Neural Network for general classification tasks

Dependent on the neural network architecture, the output vector $\hat{Y}$ can vary significantly. The output vector may pertain to probabilities that a certain example may belong to a certain classifier. In this case the normalised Soft-max activation function outputs a probability distribution such that the classifier of the training example that is highest is the most likely class for that feature vector.

As seen from 2, the ever-increasing layer-density of these models being applied to vast datasets can increase the computational complexity significantly. This further reinforces the need for fast Continual Learning and Machine Unlearning techniques to avoid the issue of naïve computationally expensive retraining operations.

Additionally, many of the techniques covered for Continual Learning and Ma-

chine Unlearning will focus on fine tuning the values of the weights and biases in order to achieve their intended effects (5).

4 Continual Learning

Continual Learning refers to the capacity of an existing machine learning model to progressively incorporate newly arriving data over time, without completely revisiting previously encountered datasets. In this setting, the model encounters batches of data, denoted mathematically as $D_{t,b} = \{X_{t,b}, Y_{t,b}\}$, where each batch b is associated with a task t. The crucial property here is that the model updates itself on $\{X_{t,b}, Y_{t,b}\}$ while ideally not depending on old datasets (6). Such an approach reduces the need to train the entire model from scratch whenever additional information arises—an attractive idea for large-scale or continuously evolving scenarios.

This incremental methodology addresses several pressing concerns. First, storing and repeatedly training on older samples can be computationally expensive. Continuous re-training from scratch may be infeasible in real-time systems or domains where data accumulates rapidly (e.g., online user interactions or streaming applications). Second, privacy considerations often dictate strict guidelines regarding data retention. Storing historical data—especially personal or sensitive information—for extended periods may run afoul of legal frameworks such as GDPR (7).

At the same time, balancing how the model retains old knowledge (stability) with how it acquires new knowledge (plasticity) is pivotal. Failing to preserve earlier distributions can lead to catastrophic forgetting, in which previously mastered information is effectively overwritten. Conversely, focusing too rigidly on old distributions can impair the model's ability to learn new patterns. Therefore, a healthy balance between stability and plasticity ensures that the model continues to learn in a dynamic environment without erasing its prior expertise.

In the following sections, we provide a taxonomy on the various continuous learning scenarios that have been explored within the recent literature. We also outline five main approaches proposed in the literature to handle Continual Learning under these constraints. However, it should be noted that while we provide an overview of all five methods for completeness, our work narrows its focus to replay-based and optimisation-based techniques. We find these two approaches the most intuitive

and best suited to our constraints: replay methods excel at mitigating catastrophic forgetting through selective re-exposure to previous data or representations, and optimisation-based techniques allow for fine-grained control of parameter updates. Despite not employing the other three approaches in our implementation, we believe describing them here helps contextualise the broader spectrum of solutions available in Continual Learning research. Finally, we provide a detailed overview on the continuous learning algorithms that we decided to implement as part of the exploratory section of our project. These algorithms are also leveraged as part of a Continuous Learning-Machine Unlearning system and manipulated to become their own unlearning algorithms which will be outlined in Chapter 6.

4.1 Continual Learning Scenarios

Continual Learning research recognises several standard experimental scenarios. Each scenario specifies what information the model receives during training and testing and how label sets or input distributions shift over time. Although all scenarios share the goal of absorbing new information without catastrophic forgetting, their practical constraints and evaluation rules differ markedly. A clear taxonomy therefore helps researchers select benchmarks that expose the strengths or weaknesses of different algorithms.

Instance Incremental Learning (IIL). All data belong to a single task while samples arrive in successive batches. The challenge is purely one of memory efficiency and online optimisation because no task boundaries or label shifts occur.

Domain Incremental Learning (DIL). Tasks share the same label set, yet each task possesses a distinct input distribution. A typical example is day images versus night images of identical object classes. The learner is never told which domain a test sample originates from, so it must generalise across domains.

Task Incremental Learning (TIL). Every task introduces a disjoint label set, and the system is informed of the active task during both training and testing. This auxiliary metadata simplifies inference and makes TIL the most widely studied benchmark suite; many foundational algorithms such as Elastic Weight Consolidation, Learning Without Forgetting, and most replay methods were first evaluated in this environment (1).

Class Incremental Learning (CIL). Successive tasks again present disjoint label sets, but the task identifier is withheld at test time. The model must therefore perform joint classification across all classes encountered so far, which is distinctly harder than TIL.

Task Free Continual Learning (TFCL). No task identifiers are provided at any stage; the learner must implicitly detect boundaries. TFCL mirrors realistic unstructured data streams but remains under explored due to its difficulty.

Online Continual Learning (OCL). Samples are observed exactly once in a single pass data stream and are never revisited. Memory and compute budgets are usually constrained, making OCL suitable for embedded or edge devices.

Blurred Boundary Continual Learning (BBCL). Label spaces overlap across tasks, so a clear notion of previous versus current knowledge disappears. Algorithms must disentangle mixed signals while still preventing forgetting.

Continual Pre training (CPT). Large foundation models receive sequential corpora during pre training with the aim of improving transfer to many future downstream tasks.

Why We Focus on Task Incremental Learning

This project concentrates on the Task Incremental scenario. Task Incremental offers three principal advantages. First, it is the most thoroughly explored setting in the literature, providing a rich collection of baselines for comparison. Second, it is challenging enough to demonstrate catastrophic forgetting, yet does not impose the additional ambiguities present in Class Incremental or Task Free settings where the learner must infer task identity at test time. Third, the explicit task identifier enables clear delineation of which parameters or gradients should be forgotten, which is essential for studying Machine Unlearning. By narrowing our attention to Task Incremental, we can evaluate the interaction between replay based and optimisation based Continual Learning strategies and recently proposed unlearning objectives without confounding factors related to hidden task information or overlapping label sets.

4.2 Continual Learning Approaches

The literature offers several broad strategies for mitigating catastrophic forgetting. *Regularisation* protects earlier knowledge by adding penalty terms that discourage disruptive parameter changes; *replay* refreshes past experience by re-introducing stored or synthetically generated samples; *optimisation* reshapes gradient updates so that new information is absorbed with minimal interference; *representation* learning aims to build general features that smoothly transfer across tasks; and *architecture* methods allocate dedicated network capacity to successive tasks. These five categories form the conceptual toolkit from which most modern algorithms are built. In the following subsections we outline each idea to provide context, even though our experimental work will later focus on a selected subset rather than the entire spectrum.

Regularisation Based Approach

Regularisation methods modify the loss function by adding terms that act as constraints or penalties. The idea behind these methods is to discourage the model from making drastic changes to certain parts of the network crucial for the performance of previously learned tasks. Within this family of approaches, there are two main ways to implement regularisation: *weight regularisation* and *function regularisation.*

Weight Regularisation. This focuses on the model's parameters, such as the weights of a neural network. The central idea is to constrain how much these parameters can change when learning a new task, effectively preserving the "knowledge" gained on earlier tasks. A well-known technique under weight regularisation is Elastic Weight Consolidation (EWC), which uses the Fisher Information Matrix to quantify how important each parameter is for the previous tasks(8). More important parameters incur a stronger penalty if changed.

Function Regularisation. By contrast, this technique aims to constrain the model's outputs or predictions rather than its internal parameters. The goal is to ensure that predictions for old tasks remain consistent even after learning new tasks. Learning Without Forgetting (LwF) is a well-known function regularisation method that stores the predicted outputs for an earlier task and includes a penalty term

to keep those predictions close to their original values (9). This helps the model maintain performance on old tasks while adapting to newer ones.

Replay-Based Approach

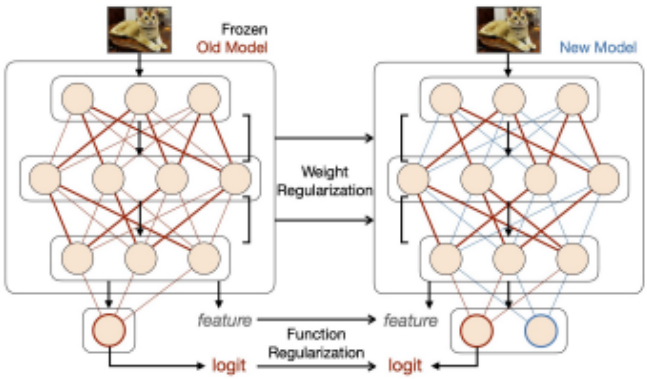


Figure 3: Visual depiction of weight regularisation and function regularisation, This direction is characterised by adding explicit regularisation terms to mimic the parameters (weight regularisation) or behaviours (function regularisation) of the old model. (1)

Replay-based approaches re-expose examples from older tasks into the training set of newer tasks. This means that the model is relearning older tasks in conjunction with learning the new task in the newest batch of data. As a result this prevents the model from solely focusing on the new task and thus helps retain knowledge of previous tasks. Similar to regularisation-based approaches there are two main replay-based approaches: Experience relay and generative replay. Experience replay works by storing a buffer of data samples from previously encountered tasks and periodically replay these samples during training on new tasks. However, as previously mentioned, this can violate GDPR (7) depending on the data we store in the buffer. An additional problem lies in buffer management, what strategies do we employ to ensure the data we store in the buffer is the best representation of a given task? The latter approach aims to tackle those caveats by training a generative model such as a Generative Adversarial Network (GAN) (10) or a Variational Autoencoder (VAE) on old task data. When the model is then learning a new task the generative model can produce synthetic examples that resemble the old task data, effectively replaying old tasks without the need to store real examples. Despite the benefits, this comes with its own challenges such as the computational

overhead and reliance on the generative model.

Optimisation-based Approach

Optimisation-Based Approaches aims to prevent catastrophic forgetting by manipulating the optimisation process used to update model parameters during learning. Instead of modifying the architecture replaying old data (as in replay-based approaches), optimisation-based approaches focus on how the learning updates (i.e., gradients) are applied to the model parameters to ensure that new tasks are learned without erasing knowledge from previous tasks. These approaches generally work by modifying the standard gradient descent process, which is the foundation of most machine learning models. In traditional gradient descent, the model parameters are updated based on the gradient of the loss function with respect to those parameters. However, in Continual Learning, simply applying gradient descent can lead to catastrophic forgetting, as the updates for new tasks can interfere with the model's performance on old tasks. Orthogonal Projection in gradient descent is an example of an optimisation-based method. The idea is to modify the gradients so that updates for a new task do not negatively affect the performance of previously earned tasks. This is done by projecting the new task's gradient onto a subspace that is orthogonal to the gradients of the old tasks.

Representation-Based Approach

Representation-Based Approaches revolve around improving the representations learned by the model to make them more robust and able to be used effectively across multiple tasks. The idea is to improve the features that the model learns in order to build a generalisable model ergo, reducing the effect of catastrophic forgetting. Representation-based approaches include self-supervised learning (SSL) and pre-training for downstream Continual Learning where the former learns representations from data without explicit labels by solving proxy tasks, for example predicting whether two images are similar. This is advantageous as models learnt via SSL tend to be more robust and less likely to overfit to a particular task. Ad-

ditionally, specific labels may not be accessible in a real world scenario so SSL allows for a way to still conduct Continual Learning with the lack of labels (11). Pre-training for downstream Continual Learning aims to initialise the model with strong generalisable representations which are then fine-tuned for specific tasks, as pre-training helps in knowledge transfer and improves the models ability to retain knowledge (12).

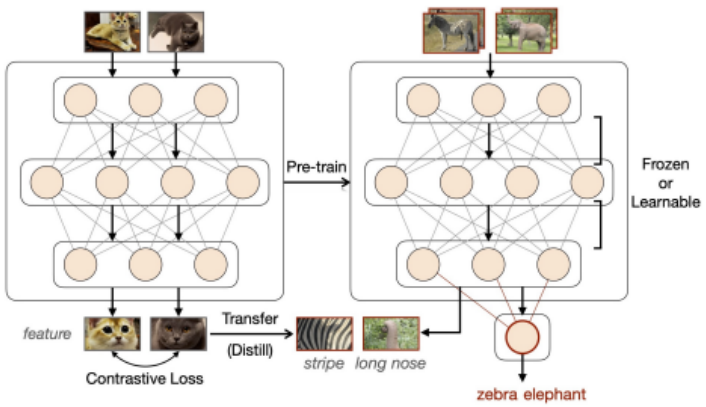


Figure 4: The representation-based approach focuses on building useful features for Continual Learning using methods like self-supervised learning and pre-training. Pre-training helps the model learn general features, and fine-tuning adjusts these features for specific tasks. The pre-trained features can either stay the same or be updated as new tasks are learned. (1)

Architecture-Based Approach

Architecture-based approaches revolve around the allocation of specific parameters for each task or using flexible architectures. Instead of having a single model shared across all tasks, architecture-based methods aim to dynamically allocate or construct task-specific parameters or sub-modules to better handle multiple tasks. This helps to avoid interference between tasks and improves knowledge retention. There are three methods in doing so. The first is parameter allocation, the simplest of the three. This is where parameters are dynamically allocated to specific tasks as they come into the system to be learned. This ensures that no task is forgotten, as they each have their own set of parameters that are only affected by other training data associated with that respective task. A problem arises as if many tasks are introduced then the architecture could run out of 'free' parameters

requiring reuse or additional constraints. Secondly, there is Model Decomposition. This separates a model into task-sharing and task-specific components, allowing more flexibility and adaptability to different tasks. By decomposing the model, Continual Learning methods can selectively share knowledge between tasks while maintaining task-specific capabilities. Finally, we could have a modular network. This uses multiple sub-networks or modules to handle different tasks. Each subnetwork is independent or loosely coupled, ensuring that learning a new task does not interfere with existing tasks. This method allows for explicit reuse of knowledge between tasks while still maintaining task-specific capabilities. It is also worth noting that the choice of basic architecture plays a crucial role in the success of continuous learning methods. Some architectural decisions, such as the width of the network or the inclusion of certain layers, can significantly impact the model's robustness to catastrophic forgetting.

4.3 Naïve Fine-Tuning

Before introducing more sophisticated continual-learning algorithms, it is essential to establish a lower-bound baseline against which all subsequent methods can be compared.

Intuition. *Naïve fine-tuning* (sometimes called *naïve rehearsal-free training*) simply continues ordinary supervised learning on each incoming task t with **only** the data of that task. No past examples are revisited; no architectural or regularisation tricks are employed to counteract catastrophic forgetting.

Algorithm 1 Naïve Fine-Tuning

Require: Datasets $\{D_1, D_2, \ldots, D_T\}$ ▷ One per task
Ensure: Final model f
  $f \leftarrow$ InitialiseModel()
  for $t \leftarrow 1$ **to** T **do**
    $f \leftarrow \textsc{Train}(f, D_t)$
  end for
  return f

Why this baseline matters. The naïve strategy is indispensable for three closely-related reasons. First, it functions as a yard-stick: if a proposed continual-learning

method cannot outperform Algorithm 1, then either the method itself or its implementation is faulty. Second, its simplicity highlights the theoretical upper bound of a model's plasticity; without constraints, weight updates can fully adapt to the current data distribution, exposing the complete extent of catastrophic forgetting. Finally, the approach is computationally and conceptually trivial—the very same single-task training loop is executed at every step—so it can be reproduced in any framework without additional hyper-parameters.

Limitations. Because earlier data is never revisited, gradient updates overwrite the representations that supported previous tasks, and empirical accuracy on those tasks can drop precipitously after only a few rounds of new learning. The model also becomes biased toward the most recent distribution, sometimes over-fitting when D_t is small. Performance is particularly unstable when the successive tasks are heterogeneous (13; 14).

When it can still be acceptable. Despite its drawbacks, plain fine-tuning may be the only viable option in settings where privacy regulations forbid storage of past data. It also suits streaming environments in which performance on the current distribution is the sole concern, and it remains invaluable as a sanity-check while debugging more sophisticated algorithms.

4.4 Naïve Replay

A slightly less constrained baseline which augments each new task with a small sample of remembered examples from earlier tasks.

Problem set-up. Suppose task k arrives as dataset $D_k = \{(x_i, t_i, y_i)\}_{i=1}^{N_k}$ where t_i identifies the task. A memory buffer $\mathcal{M}$ retains up to m examples per prior task, denoted $\mathcal{M}_k$, so that $\mathcal{M} = \bigcup_{k<t} \mathcal{M}_k$ and $|\mathcal{M}_k| = m$. However, depending on the necessity of the task(s), different strategies can be employed for memory buffer partitioning as will be discussed later (6).

Motivation. Even a handful of replayed samples typically yields a sizeable boost in retention over Algorithm 1. Implementation costs are minimal: one merely

Algorithm 2 Naïve Replay

Require: Datasets $\{D_1, D_2, \ldots, D_T\}$, memory size m
Ensure: Final model f
  $f \leftarrow \textsc{InitialiseModel}()$; $\mathcal{M} \leftarrow \varnothing$
  for $t \leftarrow 1$ **to** T **do**
    $\mathcal{M}_t \leftarrow \textsc{RandomSample}(D_t, m)$
    $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}_t$
    $f \leftarrow \textsc{Train}(f,\, D_t \cup \mathcal{M})$
  end for
  return f

concatenates the buffer with the current dataset during training, leaving both the optimisation routine and the model architecture untouched. Moreover, the parameter m provides an explicit dial for the memory–computation trade-off, allowing practitioners to align resource usage with application constraints.

Weaknesses. The simplicity of naïve replay brings three principal liabilities. Re-exposing the network to the same few stored items can drive memorisation, especially when the buffer is tiny relative to each task's size. Because the number of stored examples grows linearly with the number of tasks, per-task training time and RAM usage increase accordingly. Finally, naïve random sampling does not guarantee that the buffered items are representative of their originating distributions, so forgetting can persist; later sections discuss herding, core-set selection, reservoir sampling, and gradient-based constraints that mitigate this problem (8; 6; 15).

Typical use-cases. Naïve replay is well suited to academic benchmarks containing only a dozen-or-so tasks, where the buffer remains small and computational overhead modest. It also serves as a robust reference point when evaluating replay-centric algorithms such as GEM or AGEM, which aim to match or exceed its performance while employing significantly smaller memories. Lastly, in practical deployments where any amount of forgetting is unacceptable and resources permit, naïve replay can be an effective stop-gap until more advanced strategies are implemented.

4.5 Empirical illustration on split CIFAR-10

Protocol. We follow the standard two-task split of CIFAR-10:

$$\mathcal{T}_1 = \{\texttt{airplane}, \texttt{automobile}, \texttt{bird}, \texttt{cat}, \texttt{deer}\} \tag{4.1}$$

$$\mathcal{T}_2 = \{\texttt{dog}, \texttt{frog}, \texttt{horse}, \texttt{ship}, \texttt{truck}\}. \tag{4.2}$$

Therefore, each task contains 5,000 training and 1,000 test images (one class per label). For the replay setting, we keep a reservoir buffer of 250 samples ($\approx 10\%$ of $\mathcal{T}_1$) selected uniformly at random after training on the first task.

Results. Figure 5 contrasts naïve retraining with naïve retraining with a replay buffer. As expected, continuing to optimise only on $\mathcal{T}_2$ catastrophically degrades accuracy on $\mathcal{T}_1$ (75%→39%). Replaying just **10%** of past examples more than halves this forgetting (75%→64%) while leaving performance on $\mathcal{T}_2$ essentially unaffected.

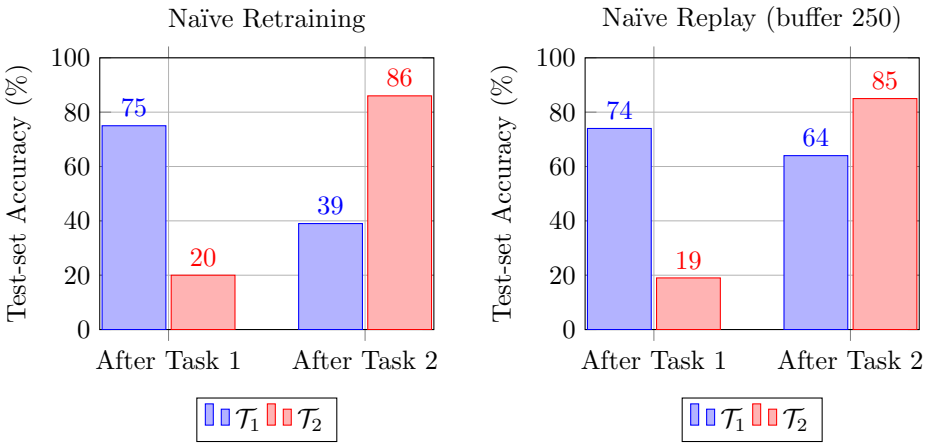


Figure 5: Test-set accuracy on the two-task CIFAR-10 split. Replaying a *small* buffer of earlier samples markedly mitigates forgetting of $\mathcal{T}_1$ while preserving plasticity on $\mathcal{T}_2$.

As demonstrated in Algorithm 2, the naïve replay method effectively retains a substantial portion of knowledge from the initial task while accommodating the learning of a subsequent task. Specifically, the model maintains 74% accuracy on the first task during the learning of the second task, in contrast to only 39% accuracy when employing naïve retraining. This result illustrates that even a relatively small buffer of previously encountered examples can significantly alleviate catastrophic forgetting. However, this benefit is accompanied by a minor reduction in performance on the second task, with accuracy decreasing from 86% to 85%.

This trade-off between knowledge retention and adaptability to new information is a well-documented challenge in the field of Continual Learning and will be further

examined in subsequent sections. Analogously, this phenomenon can be related to neuroplasticity in biological systems, wherein the brain demonstrates an ability to integrate new knowledge while preserving prior experiences (16; 6). Nonetheless, it is also known that, with age and accumulated experience, the brain exhibits a reduced capacity for acquiring new knowledge, due to the progressive rigidity of synaptic connections(16).

Although a 1% decrease in performance on a new task may appear negligible, the impact becomes more significant as the number of tasks increases and additional constraints are imposed on the learning process to mitigate forgetting. This cumulative effect underscores the importance of balancing plasticity and stability in the design of Continual Learning algorithms (6).

4.6 GEM: Gradient Episodic Memory

Gradient Episodic Memory also known as GEM (6) is the work of David Lopez-Paz & Marc'Aurelio Ranzato and is the first optimisation based approach we explore. Similar to our naïve replay method outlined in the previous section, GEM also uses a replay buffer to store samples of previously learnt tasks (memories). However, the objective of GEM is to leverage the memories to incentivise the direction that reduces loss on the new task, with theoretical guarantees of not worsening the already learnt task, while ensuring we are not directly overfitting to the samples stored within our memory buffer - something which occurs in the naïve replay based method.

Let $\mathcal{M}_k$, represent our episodic memory which stores a subset of the samples from our observed task k. Note that a sample is represented as a triplet, (x_i, t_i, y_i) which are the sample data, task identifier, and sample label respectively. In practical implementations, the algorithm is constrained by a fixed memory budget of M slots. When the total number of tasks T is known in advance, a straightforward policy is to reserve an equal share to each task, so that every task receives $m = \mathcal{M}/T$ exemplars. If the stream length is unpredictable, the reserve per task can instead be updated on the fly, progressively shrinking m whenever a new task appears (6).

In the experiments presented in this dissertation, the situation is simpler: the task sequence length T is predetermined. As a result, the per-task quota m is

fixed from the outset and remains unchanged for all subsequent analyses of GEM as well as in the later chapters where GEM is combined with unlearning techniques. This constant allocation removes the bookkeeping required in the dynamic-budget setting and lets us focus purely on the interaction between gradient constraints and forgetting behaviour.

In GEM, our predictor f_θ is parametrised by $\theta \in \mathbf{R}^p$. We define loss at the memories of the k-th task as:

$$\ell\big(f_\theta, \mathcal{M}_k\big) \;=\; \frac{1}{|\mathcal{M}_k|} \sum_{(x_i,\, k,\, y_i) \in \mathcal{M}_k} \ell\big(f_\theta(x_i, k),\, y_i\big). \tag{4.3}$$

If we were to minimise the loss of the current memory sample in conjunction with (4.3) then we would be overfitting to the memory buffer. Instead GEM uses the losses (4.3) as inequality constraints. When observing the triplet (x, t, y), GEM solves the following problem:

$$\begin{aligned} \underset{\theta}{\text{minimise}} \quad & \ell\big(f_\theta(x, t),\, y\big) \\ \text{subject to} \quad & \ell\big(f_\theta, \mathcal{M}_k\big) \;\le\; \ell\big(f_\theta^{\,t-1}, \mathcal{M}_k\big) \quad \text{for all } k < t. \end{aligned} \tag{4.4}$$

where f_θ^{t-1} is the predictor state after learning task $t-1$.

The authors of (17) note two simplifications. First, old parameter copies are unnecessary; we only need to ensure that each update does not raise the stored loss on earlier tasks. Second, under a locally linear loss and a representative memory buffer, this "no-increase" condition is equivalent to requiring the proposed update to form a non-acute angle with every past-task gradient. Hence the constraints in (4.4) reduce to simple inner-product tests, enabling the GEM projection step. Formally, the constraints are written as:

$$\big\langle g,\; g_k \big\rangle \;=\; \Big\langle \frac{\partial \ell\big(f_\theta(x, t),\, y\big)}{\partial \theta},\; \frac{\partial \ell\big(f_\theta, \mathcal{M}_k\big)}{\partial \theta} \Big\rangle \;\ge\; 0, \quad \text{for all } k < t. \tag{4.5}$$

If there are no violations of (4.5) we then have a theoretical guarantee (or very likely depending on how well the memory buffer represents the tasks learnt respectively) that our current parameter update will not increase the loss on the previous tasks. However, if there is a violation, we project the proposed gradient, g to the closest gradient in the squared ℓ^2-norm that satisfies all of the constraints, $\tilde{g}$. Thus,

we are left with a Quadratic Program (QP) :

$$\begin{aligned} \underset{\tilde{g}}{\text{minimise}} \quad & \tfrac{1}{2}\,\|g-\tilde{g}\|_2^2 \\ \text{subject to} \quad & \langle \tilde{g},\, g_k \rangle \;\geq\; 0, \qquad \text{for all } k < t. \end{aligned} \tag{4.6}$$

However, Since the parameters of predictors in practice usually are significantly large, this QP is extremely computationally heavy. As an optimisation, we can solve the dual of this QP:

$$\begin{aligned} \underset{v}{\text{minimise}} \quad & \tfrac{1}{2}\, v^\top G G^\top v + g^\top G^\top v \\ \text{subject to} \quad & v \;\geq\; 0. \end{aligned} \tag{4.7}$$

Where, $G = (g_1, ..., g_{t-1})$. Once the Dual, (4.7) and we have found the optimal v^*, the projected gradient can be recovered by $\tilde{g} = G^T v^* + g$.

Weaknesses

Despite no longer overfitting to the memory buffer - GEM still suffers from the fact it relies upon a memory buffer. More specifically, the GDPR issues we discussed in 4.2 depending on the practical scenario (7). Furthermore, the high overhead produced from solving the Dual results in a large amount of computational resources being used compared to some other algorithms, this is further heightened by the fact solving quadratic programs are a CPU-Bound operation — much slower than GPU-Bound operations that the models operate on. And finally, there are cases whereby the model struggles at learning a new task due to the restricted search space produced by the constraints in the QP; if the projection is too far away from the proposed, the new gradient may not learn the new task.

4.7 AGEM: Average-Gradient Episodic Memory

Having established how **GEM** mitigates forgetting by projecting each update onto a cone that does not increase the loss of *any* earlier task, we can now introduce its lightweight successor, **Average Gradient Episodic Memory** or more commonly reffered to as AGEM (15). Recall that GEM must compute a separate reference gradient for every stored task and then solve a small quadratic program to find the

feasible projection. Although that procedure is tractable on modest benchmarks, it becomes a clear bottleneck as either the number of tasks or the network size grows.

AGEM keeps the same guiding principle—prevent negative interference with past experience—but replaces the many constraints of GEM with a *single* surrogate constraint. GEM ensures that the loss of each previous tasks approximated by the memory buffer does not increase. AGEM differs by instead ensuring that the average loss produced by our memories does not increase — thus leaving us with one constraint. This leaves us with the following QP:

$$\underset{\tilde{g}}{\text{minimise}} \quad \tfrac{1}{2}\,\|g-\tilde{g}\|_2^2 \quad \text{s.t.} \quad \tilde{g}^\top g_{\text{ref}} \;\geq\; 0. \tag{4.8}$$

where g_{ref} is a gradient computed using a batch randomly sampled from the episodic memory. In fact, since the QP would have at most one constraint, it is very fast to solve as we can calculate exactly how to project it to the optimal. Formally, if g violates the constraint present in (4.8) then:

$$\tilde{g} \;=\; g \;-\; \frac{g^\top g_{\text{ref}}}{g_{\text{ref}}^\top g_{\text{ref}}}\, g_{\text{ref}}. \tag{4.9}$$

The proof of this is outlined in the appendix of (15) This means that AGEM eliminates both the task–wise gradient loop and the quadratic programme, cutting time and memory overhead by roughly an order of magnitude while empirically retaining GEM's resistance to forgetting in the single–epoch regime. Despite this, GEM is still a more rigorous algorithm with it having better guarantees opposed to AGEM's performance.

4.8 Reference Metrics

Although our final assessment relies on a bespoke *CL-ToW score* (defined in Chapter 7.1), three standard continual-learning metrics provide essential context. They expose the stability–plasticity trade-offs that any combined continual-learning *and* Machine Unlearning (CL–MU) system must manage.

Let

$$R_{i,j} \in [0,1], \qquad 1 \leq j \leq i \leq T,$$

be the test accuracy on task j immediately after training task i. The upper-

triangular matrix R is computed after each training phase.

Backward Transfer (BWT) $\text{BWT} = \dfrac{1}{T-1}\displaystyle\sum_{i=2}^{T}\sum_{j=1}^{i-1}\dfrac{R_{i,j}-R_{j,j}}{T-1}$. Positive when later learning helps earlier tasks; negative values indicate forgetting.

Average Forgetting Measure (FM) $\text{F} = \dfrac{1}{T-1}\displaystyle\sum_{j=1}^{T-1}\big(\max_{i>j} R_{i,j} - R_{T,j}\big)$. Quantifies how much each task's accuracy drops from its peak.

Forward Transfer (FWT) Let $\tilde{R}_{i,j}$ be the accuracy on task j *before* it is trained. With b_j the accuracy of a randomly initialised model: $\text{FWT} = \dfrac{1}{T-1}\displaystyle\sum_{j=2}^{T}(\tilde{R}_{j-1,j} - b_j)$. Captures knowledge transfer to future tasks (6).

Role in this work. We do *not* optimise our system directly for BWT, FM, or FWT, nor do we report them as headline results. Instead, they act as diagnostic baselines: analysing how our learner scores on these conventional axes clarifies where it sits on the stability–plasticity spectrum and therefore motivates the design of the *combined* CL–ToW metric introduced in Chapter 7.1. In short, these measures frame the discussion, while the new metric ultimately drives our evaluation.

5 Machine Unlearning

Conventionally, the objective of state-of-the-art machine learning algorithms is to minimise the output value of a loss function with respect to the ground truths and the outputs of the given model, improving prediction accuracy on a given dataset. The inherent goal of Machine Unlearning is to achieve the exact opposite of this. It involves selectively increasing the loss associated with a subset of training data, with the goal of removing its influence from the model (18). The challenge lies in completing this successfully without retraining a model from scratch.

Machine Unlearning has several relevant motivations. It can be used for bias mitigation, where skewed data is removed to improve fairness and reduce model reliance on problematic correlations (19). Second, it plays a critical role in user privacy, particularly in contexts where data must be erased to comply with legal or ethical requirements (7). In such cases, the aim is to approximate the behaviour of a model trained without the sensitive data. Third, unlearning can help resolve

confusion introduced by mislabelled samples, enabling the model to recover from erroneous training inputs (20).

Importantly, recent work has highlighted that the definition of forgetting in the context of unlearning is not universal, but rather application-dependent. As introduced by Kurmanji et al., different unlearning scenarios come with distinct goals and trade-offs (4). For example, in privacy-critical settings where users request data deletion, the priority is to guarantee removal, even at the cost of some model utility (3). In contrast, for tasks like removing outdated data to keep a model up-to-date, preserving overall performance may be more important than strict deletion, and leaving minor residual traces may be acceptable. These differing priorities shape the choice of unlearning strategies and evaluation criteria.

5.1 Terminology

Before discussing existing algorithms, we first introduce commonly used terminology and evaluation metrics to clarify the objectives of Machine Unlearning and define what constitutes a successful unlearning outcome.

To formally define the Machine Unlearning problem, we adopt the notation and framework introduced by Zhao et al.(21). Let $\mathcal{D}_{\text{train}}$ denote the full training dataset, which can be partitioned into a *forget set* $\mathcal{S}$ and a *retain set* $\mathcal{R} = \mathcal{D}_{\text{train}} \setminus \mathcal{S}$. A model is initially trained using an optimisation algorithm $\mathcal{A}$, such that the resulting parameters are given by:

$$\theta^{\text{o}} = \mathcal{A}(\mathcal{D}_{\text{train}}).$$

The goal of an unlearning algorithm $\mathcal{U}$ is to produce an updated model θ^{u} by removing the influence of the forget set $\mathcal{S}$ from the trained model θ^{o} while preserving knowledge from the retain set. Formally, this is expressed as:

$$\theta^{\text{u}} = \mathcal{U}(\theta^{\text{o}}, \mathcal{S}, \mathcal{R}).$$

An ideal unlearning algorithm produces a model θ^{u} that closely approximates the model that would have resulted from retraining on $\mathcal{R}$ alone.

This notion is captured through the concept of (ϵ, δ)-*closeness*. Specifically, let μ and ν be distributions over model outputs induced by $\mathcal{A}(\mathcal{R})$ and $\mathcal{U}(\theta^{\text{o}}, \mathcal{S}, \mathcal{R})$,

respectively. The two models are said to be (ϵ, δ)-close if for any measurable event $\mathcal{B}$, the probability assigned to $\mathcal{B}$ by the unlearned model does not exceed that of the retrained model by more than a multiplicative factor e^{ϵ} and an additive term δ:

$$\mu(\mathcal{B}) \leq e^{\epsilon}\nu(\mathcal{B}) + \delta.$$

5.2 Evaluating Unlearning Success

Exact unlearning is achieved when the distributions over outputs induced by the original model and unlearn model are (ϵ, δ)-close with $\epsilon = \delta = 0$. However, due to the computational cost of retraining, most practical methods aim for approximate unlearning and are evaluated empirically across three core criteria: utility, efficiency and forgetting quality (4).

Utility measures performance on the retain set $\mathcal{R}$ and on a hold-out test set $\mathcal{D}_{\text{test}}$. Retain accuracy $a_{\mathcal{R}}$ and test accuracy a_{test} should remain close to those of the original model or its retrained counterpart.

Efficiency is the runtime or computational cost of applying $\mathcal{U}$ compared to retraining. This is often measured in wall-clock time or total number of parameter updates.

Forgetting quality quantifies how effectively a model removes the influence of the forget set $\mathcal{S}$. This is non-trivial to calculate directly, and several proxies have been proposed. A basic method is to measure the classification accuracy $a_{\mathcal{S}}$ of the unlearned model on the forget set. Ideally, this should approximate the accuracy obtained by retraining on the retain set $\mathcal{R}$ alone, denoted $a^{\text{r}}_{\mathcal{S}}$.

Membership Inference Attacks (MIAs)

In modern unlearning literature, a widely accepted metric for evaluating forgetting quality is a model's robustness to Membership Inference Attacks (MIAs) (4; 22; 23). These attacks aim to determine whether a particular data point was part of the training set. Inadequate unlearning may leave detectable traces, which can be exploited to infer that a sample was once present and has since been removed—undermining the privacy guarantees that unlearning seeks to provide.

Formally, given a trained model f, an input x, and a label y, an adversary

constructs a binary decision function $\phi(f(x), y) \to \{0, 1\}$, where $\phi(f(x), y) = 1$ indicates that (x, y) is predicted to be a member of the training dataset. The decision is typically based on whether a membership score (e.g., model loss or confidence) exceeds a predefined threshold. Traditional MIAs operate at the population level, applying a single scoring rule across all samples.

To better assess residual memorisation following unlearning, Kurmanji et al. propose unlearning-adaptive MIAs (4) later referred to as U-MIAs in (23). Their *Basic MIA* replaces thresholding with a supervised learning approach: the cross-entropy loss $\ell(x, y)$ is used as input to a binary classifier trained to distinguish between samples from the forget set $\mathcal{S}$ (labelled 1) and an unseen test set $\mathcal{T}$ (labelled 0):

$$\mathcal{D}_{\text{train}}^{\text{binary}} = \{(\ell(x_i, y_i), y_i^b)\}, \quad y_i^b = \begin{cases} 1 & \text{if } x_i \in \mathcal{S} \\ 0 & \text{if } x_i \in \mathcal{T} \end{cases}$$

The classifier's accuracy on a held-out evaluation set reflects how distinguishable the forget set remains. A successful unlearning defence should yield an attack accuracy near 50%. To prevent distributional mismatch, $\mathcal{T}$ is matched to the class distribution of $\mathcal{S}$.

Kurmanji et al. also introduce a stronger, unlearning-adapted version of the Likelihood Ratio Attack (LiRA) (22). In its original form, LiRA trains shadow models—some with the target sample in their training set, and others without—to estimate two distributions over model outputs. For unlearning, this idea is extended by applying the unlearning algorithm to the models trained with the sample. This yields two new distributions: one for samples that were seen and subsequently unlearned, and one for samples that were never seen. The attacker then computes a per-sample likelihood ratio:

$$\Lambda(x) = \frac{P(\phi(f(x)_y) \mid x \text{ seen and unlearned})}{P(\phi(f(x)_y) \mid x \text{ never seen})}$$

This enables fine-grained inference of whether the model has effectively forgotten a specific sample.

Building on this, Hayes et al. formalise the framework as a per-example evaluation method for unlearning, which they term U-LiRA (23) . Their results demon-

strate that even when population-level metrics suggest successful forgetting, individual forgotten samples may still be recoverable. This underscores the need for per-example U-MIAs, which provide a more granular lens for evaluating unlearning effectiveness and can uncover residual memorisation that population-level methods fail to detect.

As MIAs have become a standard for evaluating unlearning, robustness to such attacks is now a key benchmark for assessing privacy. This raises a fundamental trade-off: stronger forgetting tends to improve privacy but can reduce model utility, while preserving utility may leave models more vulnerable to MIAs. As different applications have varying privacy and performance requirements, unlearning methods must be adaptable — allowing this trade-off to be tuned based on the specific use case.

5.3 Baseline Forgetting Methods

Outlined in one of the earliest works on machine unlearning, Golatkar et al. introduce several basic techniques for forgetting a subset of training data (24). These methods, alongside *retrain* - retraining of the model on the retain set, have since become standard baselines for comparing the effectiveness of newer unlearning algorithms across metrics such as utility, forgetting quality, and resistance to membership inference attacks.

Fine-Tune

Fine-tune unlearning works by retraining the model on the retain set $\mathcal{R}$ with a slightly higher learning rate. The increased learning rate encourages catastrophic forgetting on the forget set and exploits the tendency of neural networks to quickly overwrite older representations when trained on new data.

Random Labelling (RL)

Random Labelling replaces the true labels of the forget set $\mathcal{S}$ with randomly assigned labels, and then fine-tunes the model on the full dataset. The resulting gradients for those samples are noisy and cause the model to forget their original association with the forgotten labels.

NegGrad

Negative Gradient (NegGrad) unlearning involves applying gradient ascent on the forget set $\mathcal{S}$, rather than descent, or more formally, carrying out gradient descent on the negative loss on the forget set. This increases the loss on forgotten samples and pushes the model parameters away from regions that correctly classify them.

NegGrad+

Kurmanji et al. introduced NegGrad+ as an extension of NegGrad (4). The newer algorithm incorporates both retain and forget sets during fine-tuning, applying standard gradient descent to the retain set weighted by a factor α, while simultaneously negating gradients for the forget set weighted by $(1 - \alpha)$ to improve performance on the retain set (Equation 5.1).

$$\mathcal{L}(w) = \alpha \times \frac{1}{|\mathcal{R}|} \sum_{i=1}^{|\mathcal{R}|} l(f(x_i; w), y_i) - (1 - \alpha) \times \frac{1}{|\mathcal{S}|} \sum_{j=1}^{|\mathcal{S}|} l(f(x_j; w), y_j) \tag{5.1}$$

5.4 SalUn: Saliency Unlearning

Fan et al. introduced an approach to machine unlearning by targeting the most influential model parameters responsible for memorising the forget set $\mathcal{S}$ (5). SalUn proposes that only a subset of parameters meaningfully encode the forgotten information and therefore, effective unlearning can be achieved by modifying this subset alone. By selectively avoiding updates on irrelevant parameters it aims to reduce the risk of degrading performance on the retain set.

To identify which parameters are critical for forgetting, SalUn constructs a weight saliency map by computing the gradient of the forgetting loss ℓ_f with respect to the model parameters θ, evaluated at the pre-trained model θ_o:

$$m_s = \left(\left|\nabla_\theta \ell_f(\theta; \mathcal{S})\right|\theta = \theta_o\right| \geq \gamma\right). \tag{5.2}$$

Here, $|\nabla_\theta l_f(\theta; \mathcal{S})|_{\theta=\theta_o}|$ is the magnitude of the gradient of the loss on the forget set compared to a threshold γ that determines the minimum required saliency for a parameter to be selected. This is typically set to the median of the gradient magnitudes. The function is applied element-wise, producing a binary mask m_s that

identifies which parameters are sufficiently salient to be updated during unlearning. While a soft-thresholding variant of SalUn was proposed, empirical results show that the default hard-thresholding strategy performs just as well hence making it the preferred implementation in practice.

Using the saliency map, SalUn updates only the salient weights while freezing the rest. The model parameters after unlearning, θ_u, are computed as:

$$\theta_u = \underbrace{m_s \odot (\Delta\theta + \theta_o)}_{\text{updated salient weights}} + \underbrace{(1 - m_s) \odot \theta_o}_{\text{unchanged non-salient weights}}, \tag{5.3}$$

where $\Delta\theta$ is a learned update (corresponding to an unlearning algorithm e.g. gradient ascent), $\odot$ denotes element-wise multiplication, and θ_o is the original model.

SalUn defines separate unlearning objectives tailored for classification and generation tasks, each combining a forgetting mechanism with regularisation to preserve performance on the retain set $\mathcal{R}$.

Focusing on the classification setting, SalUn builds on the Random Labeling (RL) method. It replaces the true labels in the forget set S with incorrect ones, denoted $y' \neq y$, and fine-tunes the model using a loss function with two components:

- **Forgetting term:**

  $$\mathbb{E}_{(x,y)\sim\mathcal{S},\ y'\neq y}\left[\ell_{\mathrm{CE}}(\theta_u; x, y')\right]$$

  This applies cross-entropy loss using randomly assigned labels, encouraging the model to unlearn its original associations by reducing confidence in the true labels.

- **Retention term:**

  $$\mathbb{E}_{(x,y)\sim\mathcal{R}}\left[\ell_{\mathrm{CE}}(\theta_u; x, y)\right]$$

  This regularisation term reinforces correct predictions on the retain set $\mathcal{R}$, thereby helping preserve the model's generalisation ability.

The overall objective is a weighted sum of these two terms, controlled by a hyperparameter $\alpha > 0$. A higher α places more emphasis on retention, while a lower value prioritises forgetting.

SalUn's modular nature allows it to be applied on top of existing unlearning methods, acting as an extension that restricts updates to only the most salient

weights. Empirical results demonstrate that this selective approach improves forgetting performance while also improving utility on the retain set. Notably, combining SalUn with Random Labeling (RL) consistently outperforms standalone methods across both classification and generative tasks (5).

5.5 SCRUB: Scalable Remembering and Unlearning unBound

Kurmanji et al. introduced SCRUB, a machine unlearning framework that casts forgetting as a model editing task using a teacher-student setup (4). In contrast to methods that directly manipulate model weights or rely on retraining from scratch, SCRUB trains a new student model to retain performance on the retain set $\mathcal{D}_r$ while intentionally diverging from the original (teacher) model's behaviour on the forget set $\mathcal{D}_f$.

Formally, given a teacher model $f(x; w_o)$ and a student model $f(x; w_u)$, SCRUB minimises the following contrastive objective:

$$\begin{aligned}\mathcal{L}(w_u) = \mathbb{E}_{(x,y)\in\mathcal{R}}\left[\alpha \cdot d(f(x; w_o), f(x; w_u)) + \gamma \cdot \ell(f(x; w_u), y)\right] \\ - \ \mathbb{E}_{x\in\mathcal{S}}\left[d(f(x; w_o), f(x; w_u))\right], \end{aligned} \tag{5.4}$$

where $d(\cdot, \cdot)$ denotes the KL divergence — a measure of how different the student's output distribution is from the teacher's —, and ℓ is the supervised task loss (e.g., cross-entropy). The hyper-parameters $\alpha, \gamma > 0$ control the trade-off between retention and utility.

The first two terms encourage the student model to stay close to the teacher on the retain set and maintain its predictive accuracy, while the final term pushes the model to move away from the teacher on the forget set, achieving unlearning.

In privacy-sensitive scenarios, forgetting too aggressively can distort the model's behaviour and make it easier to infer membership via MIAs. To address this, the authors propose SCRUB+R, which introduces a rewinding mechanism. During training, multiple checkpoints are saved. The final model is selected by evaluating which checkpoint most closely matches the performance of a model retrained from scratch on $\mathcal{D}_r$. This prevents overfitting to the forget set and helps match the privacy guarantees of retraining while preserving utility.

5.6 RUM: Refined-Unlearning Meta-Algorithm

Zhao et al. investigate the underlying properties of training data that influence the difficulty of unlearning (21). They identify two primary factors: memorisation and entanglement.

Memorisation measures how much a model relies on a particular training example to produce correct predictions. It was formalised by Feldman et al. as:

$$\text{mem}(\mathcal{A}, \mathcal{D}, i) = \Pr_{f \sim \mathcal{A}(\mathcal{D})}[f(x_i) = y_i] - \Pr_{f \sim \mathcal{A}(\mathcal{D} \setminus i)}[f(x_i) = y_i], \tag{5.5}$$

where $\mathcal{A}$ is a training algorithm, $\mathcal{D}$ the training dataset, and (x_i, y_i) the input-label pair for example i (25). This score captures the predictive gap caused by removing i from the dataset. Higher memorisation scores suggest greater dependence on that example.

Entanglement quantifies how intertwined the forget set S and the retain set $\mathcal{R}$ are in the learned representation space. If these sets are highly entangled, unlearning one can easily interfere with the other. The entanglement score (ES) is computed using the spread and distance of representations from their centroids (26):

$$\text{ES}(\mathcal{R}, \mathcal{S}; \theta_o) = \frac{\frac{1}{|\mathcal{R}|} \sum i \in \mathcal{R} |\phi_i - \mu_R|^2 + \frac{1}{|\mathcal{S}|} \sum j \in \mathcal{S} |\phi_j - \mu_S|^2}{\frac{1}{2} \left(|\mu_{\mathcal{R}} - \mu|^2 + |\mu_S - \mu|^2 \right)}, \tag{5.6}$$

where $\phi_i = g(x_i; \theta_o)$ is the embedding of x_i under the model, and $\mu_{\mathcal{R}}$, μ_S, and μ are the centroids of the retain set, forget set, and full dataset respectively.

A key empirical observation made is that different unlearning methods perform best on forget sets with different memorisation characteristics. For example, low-memorised examples often require no intervention, and are naturally forgotten due to catastrophic forgetting. Hence, applying explicit unlearning can lead to over forgetting and open up susceptibility to MIAs. RUM exploits this characteristic using the following two steps:

Refinement

Real-world forget sets tend to contain a heterogeneous mix of memorisation levels. Hence the first step is to **refine** the training data. The forget set $\mathcal{S}$ is partitioned into K disjoint subsets $\{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_K\}$ based on each example's memorisation score.

A common setup involves three groups: low-, medium-, and high-memorisation.

Meta-Unlearning

RUM then defines a meta-algorithm M that performs unlearning sequentially over the refined subsets. This consists of Algorithm Selection $M_U(\mathcal{S}_i)$ and Execution Ordering $M_O(\{\mathcal{S}_i\})$.

- $M_U(\mathcal{S}_i)$: selects an unlearning method $U_i \in \{U_1, ..., U_N\}$ for each subset $\mathcal{S}_i$, based on its memorisation profile. The selection is informed by empirical insights.
- $M_O(\{\mathcal{S}_i\})$: determines the order of execution. A common strategy is to apply unlearning from low to high memorisation ($\mathcal{S}_{\text{low}} \to \mathcal{S}_{\text{medium}} \to \mathcal{S}_{\text{high}}$).

At each step i, RUM applies U_i to subset $\mathcal{S}'_i$, using as the retain set:

$$\mathcal{R}_i = \mathcal{R} \cup \{\mathcal{S}'_j \mid j > i\}$$

This ensures that unlearning each subset does not harm the remaining forget subsets. The process returns the final model θ_u^K after all subsets have been sequentially unlearned.

To evaluate the quality of unlearning, the paper introduces the Tug-of-War (ToW) metric, designed to assess how well a method balances the competing goals of forgetting, retention, and generalisation.

The Tug-of-War score is defined as:

$$\text{ToW}(\theta_u, \theta_r, \mathcal{S}, \mathcal{R}, \mathcal{D}_{\text{test}}) = (1-d_a(\theta_u, \theta_r, \mathcal{S}))\cdot(1-d_a(\theta_u, \theta_r, \mathcal{R}))\cdot(1-d_a(\theta_u, \theta_r, \mathcal{D}_{\text{test}})), \tag{5.7}$$

where $d_a(\theta_u, \theta_r, \mathcal{D})$ denotes the accuracy drop of model θ_u relative to the reference model θ_r on dataset $\mathcal{D}$.

A higher ToW score indicates that the unlearned model behaves similarly to the reference model across all three datasets — forgetting what it should while maintaining performance on what it shouldn't forget. This makes ToW a holistic and sensitive measure of unlearning quality.

Using ToW and MIA gap as the test metrics, experiments conducted on CIFAR-10 (27) (ResNet-18) and CIFAR-100 (27) (ResNet-50) showed that applying unlearning sequentially to memorisation-based subsets — including using the same algorithm across all subsets (referred to as RUM^F) — consistently outperformed unlearning the forget set in a single step or using randomly ordered partitions. This highlights the benefit of memorisation-aware ordering alone. The best results, however, came from Full RUM, which tailored the unlearning algorithm to each subset. On CIFAR-10, using Nothing for low-, Fine-Tune for medium-, and SalUn for high-memorised examples was the best performing configuration. On CIFAR-100, replacing SalUn (5) with NegGrad+, which is known to be the best algorithm on the larger dataset, led to the highest performance.

These results highlight that effective unlearning requires tailoring the strategy to the data — both in terms of what is being forgotten and what needs to be preserved.

6 Exploring the Interplays

This chapter assembles the learning and unlearning routines that form our CL–MU system. Each method builds on a well-studied Continual Learning baseline and adapts it—by gradient manipulation, label perturbation, or memory design—to satisfy Machine Unlearning demands.

- **NegGEM** (Section 6.1) Extends Gradient Episodic Memory by *negating* the forget-task gradient and re-projecting it, guaranteeing targeted forgetting while preserving all other tasks.
- **Unlearning-AGEM** Trades GEM's quadratic programme for AGEM's single-constraint projection, retaining efficiency without losing MU guarantees.
- **Random-Labelling GEM/AGEM** Replaces true labels with random permutations to drive confidence to chance level, enhancing resistance to membership-inference attacks.
- **GEM$^+$** Combines standard GEM for learning with NegGrad$^+$ for unlearning, serving as a "no-synergy" baseline against which integrated methods are compared.

- **ALT-NegGEM** Adds orthogonality safeguards to NegGEM to prevent over-projection as the task count grows.
- **SalUn Extension** Filters gradients through a saliency mask so that unlearning perturbs only parameters most responsible for the forget-task.
- **Dual Memory Buffer** Splits episodic storage into *learn* and *unlearn* partitions based on memorisation scores, aligning replay with CL needs and unlearning with MU objectives.

Throughout this section we highlight how each variant balances three competing axes: computational cost, retention of non-forgotten tasks, and effectiveness of erasure. The empirical consequences of these design choices are presented in Chapters 7.5 and 7.1, where the methods are benchmarked and scored using the evaluation protocol introduced earlier.

6.1 NegGEM: Negative-GEM

It has been proven that by the use of the GEM constraint (6), we can can achieve positive backward transfer on previously learned tasks while learning a new task. We leverage GEM and use this formulation to tackle the problem of machine unlearning, we call this Negative-GEM (abbreviated to NegGEM). The main rationale behind NegGEM is that we can also achieve positive backward transfer while unlearning a previously learned task. That is to say, for a specific task T_i, we want to achieve catastrophic forgetting whilst for the other tasks T_j, $(j \neq i)$, we want to achieve at least non-negative backward transfer. This is the main motivator behind our state of the art method "NegGEM".

6.1.1 Original GEM Gradient-Projection Argument

Recall from the original GEM paper (6) that we use an episodic memory for each of the old tasks $k = 1, \ldots, t-1$. If we let

$$g = \nabla_\theta L\left(f_\theta(x, t), y\right) \tag{6.1}$$

be the proposed gradient for the new task t, and

$$g_k = \nabla_\theta L\left(f_\theta, M_k\right) \tag{6.2}$$

be the gradient of the loss evaluated on the memory of each old task $k \in \{1, \ldots, t-1\}$, then the GEM insight is to prevent catastrophic forgetting by ensuring

$$\langle g_{new}, g_k \rangle \geq 0 \quad \forall k \in \{1, \ldots, t-1\} \tag{6.3}$$

Where g_{new} is the new final updated needed after any projections. In other words:

> *If the angle within the search space between g_{new} and each g_k is non-negative, then small local steps in the direction of g_{new} with gradient descent will not increase the loss on any of the old tasks.*

Local linearity assumption. Throughout this work we adopt the standard first-order (locally linear) approximation of each task loss around the current parameters θ:

$$\mathcal{L}_k(\theta + \Delta\theta) \;\approx\; \mathcal{L}_k(\theta) \;+\; g_k^\top \Delta\theta, \qquad g_k \;=\; \nabla_\theta \mathcal{L}_k(\theta).$$

Within a sufficiently small neighbourhood of θ, the loss therefore behaves like an *affine* function whose slope is g_k. Under this assumption the condition $\langle g_k, g_{\text{new}} \rangle \geq 0$ for every stored task k guarantees that an infinitesimal gradient-descent step $\Delta\theta = -\eta\, g_{\text{new}}$ (with $\eta > 0$) will *not increase* any of the past losses, because their directional derivatives along g_{new} are non-positive.

6.1.2 The Key Modification for "NegGEM"

Suppose in addition to wanting to retain all of the old tasks, There is now an older task τ that we now wish to forget. Denote:

- g_τ as the gradient of the loss evaluated on the memory of the task we want to forget
- g_k for $k \neq \tau$ as the gradient of the loss evaluated on the memory of the tasks we want to retain

Then our key goal is to move in the direction of the gradient that increases the loss on task τ, while still ensuring that the loss on all other tasks $k \neq \tau$ is not increased.

6.1.3 Negating the Task-τ Gradient

To achieve this, one simply negates the gradient of the task we want to forget:

$$g_{unlearn} = -g_\tau, \tag{6.4}$$

that is, the negative of the normal gradient for task τ. Recall that if g_τ points in the direction that reduces task-τ loss, then $-g_\tau$ points in the direction that increases task-τ loss. The initial idea was inspired by the NegGrad+ method (21), which simply adds the negative of the gradient of the task we want to forget to the gradient of the retain tasks gradients scaled by a hyperparameter α.

6.1.4 Imposing the GEM Constraint on All Other Tasks

Next, exactly as in GEM, we must ensure that the losses for the tasks we want to retain do not go up. Let

$$K = \{k \in \{1, \ldots, t-1\} | k \neq \tau\} \tag{6.5}$$

be the set of old tasks we want to preserve. To ensure no forgetting on these tasks, we impose the usual GEM constraint:

$$\langle g_{new}, g_k \rangle \geq 0 \quad \forall k \in K \tag{6.6}$$

These constraints are identical to the ones GEM imposes, except that we omit the task we want to forget τ from the set of tasks we want to preserve. In other words, we do not demand that $\langle g_{new}, g_\tau \rangle \geq 0$, we actually want $\langle g_{new}, g_\tau \rangle < 0$ to ensure the task τ is unlearned.

6.1.5 Solving the NegGEM Quadratic Program

Putting these two pieces together, we can now write the following quadratic program for each NegGEM step:

Algorithm 3 NegGEM Step

Input: g_τ, g_k for $k \in K$
Output: g_{new}
Start with $g_{unlearn} = -g_\tau$
Check if $\langle g_{unlearn}, g_k \rangle \geq 0$ for all $k \in K$
if yes **then**
$g_{new} = g_{unlearn}$
else
In the case of a violation, just as in GEM, we solve a small quadratic program that projects g_τ to the closest vector g_{new} which satisfies:

$$\langle g_{new}, g_k \rangle \geq 0 \quad \forall k \in K \tag{6.7}$$

This is done by solving the following quadratic program:

$$\min_{g_{new}} \frac{1}{2} ||g_{unlearn} - g_{new}||^2 \textbf{ subject to } \langle g_{new}, g_k \rangle \geq 0 \quad \forall k \in K \tag{6.8}$$

This optimisation problem is typically done in the dual because there are only $|K|$ constraints, and the dual is a simple linear program.
end if
Return: g_{new}

Thus "NegGEM" obtains a new gradient g_{new} that still maximally resembles the direction of g_τ needed to forget task τ, but is now also guaranteed to not increase the loss on any of the other tasks $k \in K$.

6.1.6 Why This Preserves the Key Guarantees

1. **Targeted Forgetting of Task τ**:
   - The NegGEM step is guaranteed to increase the loss on task τ.
   - Because the base direction we picked was $-g_\tau$, and because we impose no constraint such that: $\langle g_{new}, g_\tau \rangle \geq 0$, the final projection g_{new} is guaranteed to increase the loss on task τ.
2. **No Additional Forgetting on Other Tasks**:
   - The NegGEM step is guaranteed to not increase the loss on any of the other tasks $k \in K$.
   - The original GEM proof relies on the observation that if $\langle g_{new}, g_k \rangle \geq 0$, then small local steps in the direction of g_{new} with gradient descent will not increase (to first order). Because we keep all of those constraints in place for tasks $k \neq \tau$, we still retain the usual GEM guarantee: the new update cannot cause forgetting on tasks we want to retain.

Hence, the same geometric argument that GEM uses (6) applies here. The new NegGEM direction is as close as possible (in the Euclidean sense) to the forgetting direction $-g_\tau$, while still satisfying the GEM constraints for all other tasks $k \in K$.

6.1.7 Summary

Claim. Let g_τ be the gradient of the loss evaluated on the memory of the task we want to forget. Then the update g_{new} from the NegGEM QP.

$$g_{new} = \arg\min_{g_{new}} \frac{1}{2}||g_{unlearn} - g_{new}||^2 \textbf{ subject to } \langle g_{new}, g_k \rangle \geq 0 \quad \forall k \in K \quad (6.9)$$

satisfies:

1. $\langle g_{new}, g_\tau \rangle \approx \langle -g_\tau, g_\tau \rangle < 0$ (i.e. the step will increase loss on task τ)
2. $\langle g_{new}, g_k \rangle \geq 0$ for all $k \in K$ (i.e. the step will not increase loss on any of the other tasks)

This is precisely the combination required to achieve degradation of performance on task τ, while preserving everything else. Therefore, NegGEM unlearns task τ but obeys the GEM constraints for all other tasks.

6.2 Variations of Unlearning GEM

Building upon our foundational NegGEM approach, we now explore several variations that offer different trade-offs in terms of computational efficiency, performance, and privacy guarantees. These variants maintain the core principle of unlearning specific tasks while preserving performance on retained tasks, but employ different mechanisms to achieve this balance.

6.2.1 Unlearning AGEM

The Average Gradient Episodic Memory (AGEM) algorithm (15) was developed as a computationally efficient alternative to the original GEM. While GEM solves a quadratic program to project gradients, AGEM uses a simpler and faster projection method that operates on the average of gradients from memory samples.

Original AGEM Approach. In standard AGEM, when the proposed gradient violates the GEM constraints, the algorithm projects the gradient onto the closest point that satisfies these constraints. Specifically, for the standard continual learning setting, if we let g be the gradient for the new task and g_{ref} be the average gradient over episodic memories of previous tasks, the AGEM gradient projection is:

$$g_{new} = g - \frac{g^\top g_{ref}}{g_{ref}^\top g_{ref}} g_{ref}, \quad \text{if} \quad g^\top g_{ref} < 0 \tag{6.10}$$

This projection ensures that the gradient g_{new} does not increase the loss on previous tasks while remaining as close as possible to the original gradient g.

Adapting AGEM for Unlearning. To adapt AGEM for unlearning, we follow a similar approach to our NegGEM modification:

$$g_{unlearn} = -g_\tau \tag{6.11}$$

where g_τ is the gradient for the task we want to forget. We then compute g_{ref} as the average gradient over the memory samples for all tasks we want to retain:

$$g_{ref} = \frac{1}{|K|} \sum_{k \in K} g_k \tag{6.12}$$

Observe that if our memories are representative of the entire tasks dataset (which we assume), and if we were to use a random sample of the same size from every task, as we normally do in AGEM (15). This is a slight inefficiency however this is insignificant compared to the speed-up acquired by no longer using a quadratic program.

Thus, the Unlearning-AGEM update rule becomes:

$$g_{new} = \begin{cases} g_{unlearn}, & \text{if} \quad g_{unlearn}^\top g_{ref} \geq 0 \\ g_{unlearn} - \frac{g_{unlearn}^\top g_{ref}}{g_{ref}^\top g_{ref}} g_{ref}, & \text{otherwise} \end{cases} \tag{6.13}$$

This projection ensures that we move in a direction that increases the loss on task τ while not increasing the loss on retained tasks.

Algorithm 4 Unlearning-AGEM Step

Input: g_τ, memory samples from tasks in K
Output: g_{new}
Compute $g_{unlearn} = -g_\tau$
Compute average reference gradient: $g_{ref} = \frac{1}{|K|} \sum_{k \in K} g_k$
if $g_{unlearn}^\top g_{ref} \geq 0$ **then**
    $g_{new} = g_{unlearn}$ ▷ No constraint violation
else
    $g_{new} = g_{unlearn} - \frac{g_{unlearn}^\top g_{ref}}{g_{ref}^\top g_{ref}} g_{ref}$ ▷ Project to satisfy constraints
end if
Return: g_{new}

Empirical Performance Characteristics. Our empirical investigations (Section 7.2.1) reveal that continual learning AGEM provides significant computational advantages over GEM, particularly with limited memory samples. However, consistent with findings in the continual learning literature (15), we observed that AGEM does not exceed GEM's performance, but achieves comparable results with considerably lower computational costs when both algorithms are run with their optimal hyper parameters.

Interestingly, while average test accuracy across retained tasks remained similar between methods, we found that AGEM showed slightly lower retention performance on the earliest learned tasks compared to GEM. This suggests that the approximation introduced by using average gradients rather than solving the full quadratic program has subtle effects on the long-term stability of older tasks.

> **Claim.** The Unlearning-AGEM update g_{new} satisfies:
>
> 1. $\langle g_{new}, g_\tau \rangle < 0$ (increases loss on task τ)
> 2. $\langle g_{new}, g_{ref} \rangle \geq 0$ (does not increase average loss on retained tasks)
>
> However, unlike NegGEM, it does not guarantee $\langle g_{new}, g_k \rangle \geq 0$ for each individual $k \in K$, as it only preserves the inner product with the average gradient.

6.2.2 Random Labelling GEM & AGEM

Beyond the gradient negation approach, we investigate another strategy for unlearning: random label permutation. This approach draws inspiration from literature on privacy-preserving machine learning and offers distinct properties for unlearning (4).

Motivation. While NegGEM and Unlearning-AGEM directly maximise loss on forgotten tasks through gradient ascent, random labelling pursues a different objective: making the model's predictions on the forgotten task as random and uncertain as possible. This has particular relevance for defending against membership inference attacks, where an attacker attempts to determine whether specific data was used during model training.

Method Description. For the task τ that we want to forget, instead of using the true labels y_τ during the unlearning process, we randomly permute these labels to create $\hat{y}_\tau$. We then compute the gradient using these permuted labels:

$$g_{random} = \nabla_\theta L(f_\theta(x_\tau), \hat{y}_\tau) \tag{6.14}$$

Importantly, we do not negate this gradient, as the random labelling itself creates a gradient that tends to increase the loss on the original task. We then apply either the GEM or AGEM projection method to ensure this gradient does not increase loss on retained tasks.

Algorithm 5 Random Labelling GEM Step

Input: Memory samples from task τ to forget, memory samples from tasks in K to retain
Output: g_{new}
Generate random label permutation $\hat{y}_\tau$ for task τ samples
Compute $g_{random} = \nabla_\theta L(f_\theta(x_\tau), \hat{y}_\tau)$
Check if $g_{random}^\top g_k \geq 0$ for all $k \in K$
if yes **then**
    $g_{new} = g_{random}$
else
    Solve quadratic program:
    $g_{new} = \arg\min_g \frac{1}{2}||g_{random} - g||^2$ subject to $\langle g, g_k \rangle \geq 0 \quad \forall k \in K$
end if
Return: g_{new}

Similar to our adaptation of standard GEM to NegGEM, we can apply the same random labelling concept to the computationally efficient AGEM framework. This creates a more scalable variant that preserves the benefits of random labelling while reducing computational overhead.

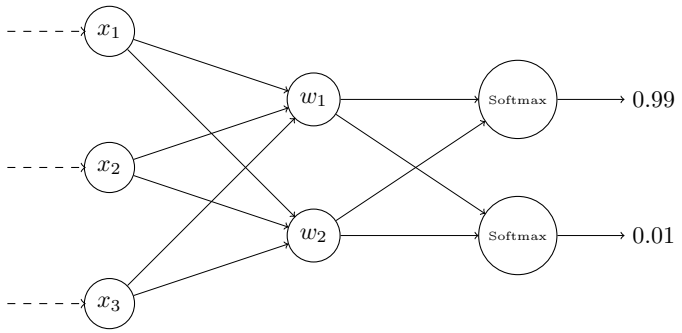
Effects on Model Predictions. A key feature of the Random Labelling variants is their effect on the model's confidence scores. Rather than simply increasing loss,

Algorithm 6 Random Labelling AGEM Step

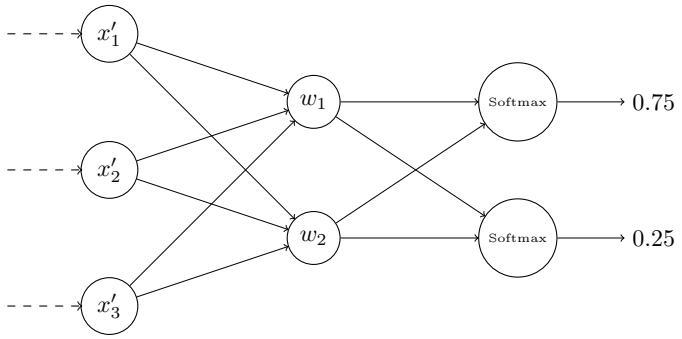
Input: Memory samples from task τ to forget, memory samples from tasks in K to retain
Output: g_{new}
Generate random label permutation $\hat{y}_\tau$ for task τ samples
Compute $g_{random} = \nabla_\theta L(f_\theta(x_\tau), \hat{y}_\tau)$
Compute average reference gradient: $g_{ref} = \frac{1}{|K|} \sum_{k \in K} g_k$
if $g_{random}^\top g_{ref} \geq 0$ **then**
    $g_{new} = g_{random}$
else
    $g_{new} = g_{random} - \frac{g_{random}^\top g_{ref}}{g_{ref}^\top g_{ref}} g_{ref}$
end if
Return: g_{new}

these methods specifically reduce the model's confidence on the forgotten task. This has direct implications for defending against membership inference attacks (MIAs).

Membership Inference Attack Vulnerability. Membership inference attacks exploit differences in model behavior on training versus non-training data. Typically, models exhibit higher confidence on data they were trained on compared to unseen data. By reducing confidence scores on forgotten data, Random Labelling approaches more effectively mitigate the risk of these attacks.



Consider a binary classification model producing prediction scores $[\hat{y}_1, \hat{y}_2]$ for two classes. If the model was trained on a sample x, it might produce confident predictions (e.g., $[0.99, 0.01]$), whereas for an unseen sample x', it might produce less certain predictions (e.g., $[0.75, 0.25]$). These confidence differentials enable attackers to distinguish between data that was or was not used in training.



The Random Labelling approaches specifically target this vulnerability by driving the model towards uncertain predictions on the forgotten task, making their training membership status harder to detect.

> **Claim.** The Random Labelling GEM and AGEM methods satisfy:
>
> 1. They reduce model confidence scores on the task to forget
> 2. They preserve the GEM/AGEM guarantees for retained tasks
> 3. They provide enhanced protection against membership inference attacks compared to gradient negation approaches

Complementary Approaches. It's worth noting that Random Labelling methods and gradient negation methods (NegGEM, Unlearning-AGEM) have complementary strengths. While gradient negation more directly maximises loss, Random Labelling more effectively reduces prediction confidence. These approaches can be combined or selected based on specific unlearning objectives and privacy requirements.

6.2.3 GEM+

So far the techniques we have discussed in this section are direct manipulations of continuous learning algorithms that can be used in an unlearning context. We hypothesise that these algorithms would perform well as there is a direct synergy between the objectives of the continual learning and machine unlearning methods, for instance, NegGEM's direct use of the same constraints. As we wish to explore the interplays between these two techniques, we thought there would be prose to combining two algorithms that do not have synergies to see if there are any innate

positive or negative attributions to one another.

This led us to creating GEM+. This algorithm is a combination of GEM and Neggrad+, that is, we use GEM to learn new tasks and Neggrad+ to handle task forget requests. We anticipate that GEM and Neggrad will still perform well together as they there are no conflicting elements between the respective algorithms. However, we hypothesise that they would underperform in contrast to the other algorithms mentioned above. This algorithm acts as our baseline combination test, we can directly compare how effectively neggrad forgets on a model that does not continually learn and one that does. We can further compare the retention accuracies of this algorithm with the other algorithms in this section to observe if careful engineering is required to get optimal results; or if combining CL and MU shows no prose.

6.2.4 ALT-NegGEM

GEM has a major drawback. Recall that GEM operates in the following manner: to learn task t we compute g_t, check if g_t is at least orthogonal to all g_k s.t. $k \in \{1 \cdots t-1\}$, if g_t satisfies this, we accept the gradient otherwise, we project g_t to the closest gradient in the L2-Norm that is orthogonal and we accept this. As the amount of tasks we have learned increases, the search space for gradients that are at least orthogonal to the learned tasks significantly decreases. As a result, our gradient projection can be significantly far from our original g_t. This is problematic as if we project too far away, our gradient no longer has the property that it learns the new task; thus the accuracy of task t will be low.

ALT-NegGEM aims to address this shortfall and add an additional constraint to ensure that this very problem cannot happen. Observe that the following shortfall will also be present in the unlearning operation of NegGEM or any other variation that uses (A)GEM in its unlearning mechanism without directly addressing the issue. We handle these cases separately in a similar fashion.

Addressing the CL shortfall: This case is trivial. We add an extra constraint to our GEM projection that ensures that our projected gradient is at least orthogonal to g_t.

Addressing the MU shortfall: Handling the Machine unlearning case is not as straightforward. This is because we wish to add a constraint that ensures that the task we wish to unlearn, g_u is at most orthogonal to the our projected gradient. However, if we introduce an inequality whereby the sign has changed, this causes the dual to change and must be formulated differently. To avoid having to do this we identified that we could simply use the negated gradient of task u. Mathematically, this is written as:

$$\langle \tilde{g},\, -g_u \rangle \geq \lambda$$

Hence, if $\lambda = 0$ then we enforce orthogonality, since we want our task to be strictly forgotten, we need $\lambda > 0$. Since the dot product represents the direction, negating the gradient of g_u is equivalent to multiplying the entire left side by negative 1. Thus this inequality can be rearranged to:

$$\langle \tilde{g},\, g_u \rangle \leq \lambda$$

Thus, this proves that the previous inequality is sound. At this point, the gradient from which we project from is unknown. We use the average gradient of the retain tasks to encourage projections large enough to escape possible local minima (now our projections can be large as they are "safe") However, using $-g_u$ is also a viable option.

6.3 SalUn Extension

Building upon the saliency-based unlearning approach proposed by Fan et al. (5), we integrate a salient weight filtering step directly into our unlearning pipeline. Specifically, before applying our chosen unlearning algorithm, we identify the most salient parameters to the forget-task (τ) by selecting the largest gradient components in absolute value. The subsequent NegGEM projection step then acts exclusively on these salient parameters. Algorithm 7 summarises this filtering procedure:

Algorithm 7 Saliency Based Gradient Filtering

Input: Gradient g_τ for forget-task τ, threshold p

Flatten gradient: grad $= g_\tau$.view(-1)

Compute cutoff threshold: cutoff $=$ Quantile$_{1-p}(|\text{grad}|)$

Create mask: mask $= |\text{grad}| \geq$ cutoff

Apply mask to gradient: grad $=$ grad $\times$ mask

return filtered gradient reshaped, mask

After applying SalUn filtering to the forget-task gradient (g_τ), the resulting sparse gradient g_τ^{sal} is negated and used as the initial direction for unlearning. The NegGEM quadratic program is then applied, projecting this sparse direction onto the feasible region defined by the constraints of retained-task performance. By explicitly isolating the salient parameters before the NegGEM optimisation step, we ensure targeted unlearning, reducing unnecessary perturbation of model parameters unrelated to the forgotten task, without compromising forgetting.

6.4 Memory Buffer Strategy

To guide the design of our memory system, we first conducted experiments measuring the effect of memorisation level on continual learning performance. Specifically, we trained models across 10 sequential tasks using three memory buffer strategies: random replay, replay of the least memorised samples, and replay of the most memorised samples.

Each buffer configuration was evaluated over three independent runs, logging the full task-accuracy matrices at each step during training. This enabled us to quantify not only the final performance, but also the degree of forgetting each task experienced, measured as the difference between its accuracy immediately after learning and its final accuracy after all tasks had been seen.

We report the mean and standard deviation of both final accuracies and forgetting across the three runs, providing a comparison between buffer strategies.

Our results show that prioritising low-memorised examples for replay during continual learning not only slightly improves final accuracy but, more notably, reduces catastrophic forgetting. While this outcome aligns with intuition — reinforcing weaker patterns strengthens long-term retention — it has received little direct

Table 1: Final accuracy and average forgetting for each memory buffer composition.

Buffer Config	Final Acc. (%)	Avg. Forget (%)
Random	65.15 ± 0.58	5.42 ± 0.61
Most Memorised	66.48 ± 1.39	4.01 ± 0.92
Least Memorised	67.95 ± 1.80	3.37 ± 0.41

investigation in existing continual learning research.

In parallel, recent studies in machine unlearning have shown that highly memorised samples are the primary contributors to retained information and must be specifically targeted for effective unlearning (21).

Based on these two complementary insights, we implemented a dual memory buffer architecture that partitions training samples by memorisation level on a *per-task* basis. This structure allows continual learning (CL) and machine unlearning (MU) to operate on the types of data most beneficial to their respective objectives.

6.4.1 Memorisation-Based Buffer Allocation

Let $\mathcal{D}_t$ denote the training data for task t, and let $m(i) \in \mathbb{R}$ represent the memorisation score for each sample $i \in \mathcal{D}_t$, as defined in Section 5.6. For a fixed buffer size $s \in \mathbb{N}$ per task, we define a tunable parameter $t \in [0, 1]$, which controls the proportion of each buffer to be filled with samples at the extremes of the memorisation distribution within that task.

Specifically, $t \cdot s$ samples are selected based on the highest or lowest memorisation scores (depending on the buffer type), and the remaining $(1-t) \cdot s$ samples are chosen uniformly at random from the rest of $\mathcal{D}_t$. This approach emphasises memorisation-critical examples while maintaining sufficient diversity to avoid overfitting or overly narrow forgetting. It also allows flexibility for both experimentation and practical use by enabling customisable buffer compositions for different scenarios.

6.4.2 Buffer Usage in CL and MU

During *Continual Learning*, the model accesses only the learn buffer $\mathcal{B}^t_{\text{learn}}$, which contains under-memorised samples for task t, helping to preserve generalisation and reduce catastrophic forgetting.

During *Machine Unlearning*, both buffers are used as input to the NegGEM update. The unlearn buffer $\mathcal{B}^t_{\text{unlearn}}$ provides highly memorised samples, serving

as primary targets for forgetting. Meanwhile, samples in the learn buffer $\mathcal{B}^t_{\text{learn}}$, though initially low-memorised, may become memorised through repeated replay during CL, making them newly relevant for unlearning.

This dual-buffer setup improves both learning and unlearning outcomes in line with prior findings, without requiring any modification to the NegGEM algorithm itself.

7 Testing

The purpose of this section is to establish *how* we measure the quality of a combined Continual Learning - Machine Unlearning (CL–MU) system, *what* baselines and datasets we use, and *which* stress–test schedules the algorithms must survive. In order:

1. **Metrics:** We begin with accuracy and confidence on *retain* vs. *forget* sets, introduce a continual-learning variant of the Tug-of-War score (CL-ToW), and adopt the basic membership-inference attack as a privacy probe. These reference metrics contextualise, but do not replace, the bespoke CL–MU score defined later in Chapter 7.1.

2. **Baselines:** Naïve Replay, GEM, and A-GEM are re-implemented from scratch to serve as pure continual-learning anchors; their unlearning extensions (NegGEM, Unlearning-AGEM, etc.) constitute the systems under test.

3. **Task-sequence suite:** Eight sequences (A–H) systematically vary the size, position, and timing of forget requests, yielding a minimal yet covering test-bed for plasticity, stability, and reversibility.

4. **Experimental setup:** CIFAR-10 and CIFAR-100 are split into tasks. Each algorithm is tuned on CIFAR-10 for a fast turnaround and then locked before moving to CIFAR-100, where the larger continuum stresses memory and projection routines.

5. **Results:** We report CL-ToW, retain/forget accuracy, confidence, wall-clock cost, MIA vulnerability and CUL_{mix}, a new metric we introduce, to evaluate

the overall performance of the models. We follow this by an ablation on saliency masking and buffer composition.

Together, these components provide a transparent and reproducible framework for judging whether a candidate method can *learn continuously, forget selectively, and do so efficiently.* All code, hyper-parameters, and raw logs are released alongside this manuscript in our GitHub repository.

7.1 Metrics

Accuracy

Accuracy metrics are measured using exact class label matches between model predictions and ground truth, and averaged over all tasks in each category. We evaluate accuracy across four different sets:

- **Retain Accuracy:** Computed on all retained tasks using their corresponding memory buffers. This measures how well the model preserves learned knowledge after unlearning.
- **Forget Accuracy:** Computed on the tasks designated for unlearning. A successful unlearning strategy should reduce this metric close to random guess level.
- **Test Accuracy (Retain):** Accuracy on test sets associated with retained tasks. This indicates generalisation performance beyond memorised data.
- **Test Accuracy (Forget):** Accuracy on test sets for forgotten tasks. Ideally, this should decrease as unlearning progresses, while ensuring that the model does not relearn forgotten patterns through other tasks.

Confidence

For each prediction, we extract the maximum softmax probability across all classes. These maximum confidence values are averaged across all samples to compute the average prediction confidence. We evaluate accuracy across the same four sets detailed in the Accuracy section.

High confidence on forgotten data suggests incomplete unlearning, while reduced confidence implies effective removal of the data's influence.

Tug-of-War (ToW)

The original Tug-of-War (ToW) metric is defined as:

$$\text{ToW}(\theta_u, \theta_r, \mathcal{S}, \mathcal{R}, \mathcal{D}_{\text{test}}) = (1-d_a(\theta_u, \theta_r, \mathcal{S}))\cdot(1-d_a(\theta_u, \theta_r, \mathcal{R}))\cdot(1-d_a(\theta_u, \theta_r, \mathcal{D}_{\text{test}}))$$

where the accuracy difference $d_a(\theta_u, \theta_r, \mathcal{D})$ is defined as the absolute magnitude between an unlearned model and a retrained model (21).

However, this formulation is not applicable to continual learning and unlearning, where retain and forget sets evolve over time. In this setting, the "retrained" baseline becomes the model that has continually learned all tasks up to time step t.

We define the forget and retain sets as:

$$\mathcal{S} = \{\mathcal{D}_i \mid 0 < i < t,\ \mathcal{G}_i < 0\}$$

$$\mathcal{R} = \{\mathcal{D}_i \mid 0 \leq i < t,\ \mathcal{G}_i > 0\} \setminus \mathcal{S}$$

The forget and retain sets change according to the task sequence step t. The task sequence is given by:

$$\text{task sequence} = \{0, \ldots, 19, -19, \ldots, -0\}$$

where positive values denote learning iterations and negative values denote unlearning that task.

We then define CL-ToW at iteration t as:

$$\text{CL-ToW}(t) = \begin{cases} (1 - d_a(\mathcal{S}_t)) \cdot (1 - d_a(\mathcal{R}_t)) \cdot (1 - d_a(\mathcal{D}_{\text{test}})), & \text{if } t > t_0 \\ 1, & \text{if } t \leq t_0 \end{cases}$$

This extends ToW to continual learning by dynamically adjusting forget and retain sets and comparing models at each step t. However, if a matching set of retain set forget set pairs does not exist at timestep t as it has not existed in a previous continual learning phase, then we have no baseline to compare to. This

can occur if we forget the task that was not the most recently observed as a continual learner. The task accuracy for the original continually learnt model with matching retain and forget set would not be present in a previous iteration. Although CL-ToW is well defined for the task sequence presented above, we require other metrics for different task sequence orderings.

CLU: a task–adaptive score for Continual–Learning × Unlearning

Why another metric? Existing measures such as backward transfer (BWT) or the tug-of-war (ToW) index rely on a full *re-trained* reference model at every time–step (21). That requirement is prohibitive when (i) the task stream is long, (ii) the forget/retain sets change online, or (iii) privacy constraints forbid re-training on the original data. We therefore introduce a lightweight scalar score, **CLU**, that:

- *penalises* loss of performance on the **retain** tasks,
- *penalises* residual knowledge on the **forget** tasks,
- dispenses with any auxiliary model or data replay,
- remains in the fixed range $[0, 1]$ for easy comparison across runs,
- supports tasks that can flip from retain to forget (or vice versa) at arbitrary time–steps.

Notation. After iteration t of a mixed learn/unlearn stream let R_t and S_t denote the current retain and forget task sets, obtained from the *last* sign of every task identifier in the sequence $\{T_0, \ldots, T_t\}$. For every task j we store once and for all the accuracy, a_j^{anchor}, reached when that task was *first* completed (anchor accuracy). $a_j(t)$ is the model's current accuracy on task j and a_j^{rand} is the chance accuracy (0.2 for the 5-way image tasks considered here).

Components.

$$L_{\text{ret}}(t) = \frac{1}{|R_t|} \sum_{j \in R_t} \max\big(0,\ a_j^{\text{anchor}} - a_j(t)\big), \tag{7.1}$$

$$L_{\text{res}}(t) = \frac{1}{|S_t|} \sum_{j \in S_t} \max\big(0,\ a_j(t) - a_j^{\text{rand}}\big). \tag{7.2}$$

L_{ret} measures *how much* the learner has forgotten of the tasks it must keep; L_{res} measures how far its predictions on the tasks it should forget still exceed random guessing.

Definition. We define

$$\boxed{\text{CLU}(t) = \big(1 - L_{\text{ret}}(t)\big)\,\big(1 - L_{\text{res}}(t)\big)} \quad \in [0, 1].$$

Interpretation and properties.

- **Perfect behaviour** ($L_{\text{ret}} = L_{\text{res}} = 0$) gives CLU = 1.
- If either retained accuracy collapses or residual accuracy stays near chance, the corresponding factor drops towards zero and pulls the product down rapidly (worst-component dominance).
- Multiplicativity ensures symmetry: the metric reacts equally to a 5 % rise in forgetting or a 5 % fall in retention.
- Because only anchor accuracies and per-task chance levels are needed, CLU is *model-independent* and incurs $\mathcal{O}(|R_t| + |S_t|)$ work per step.
- Tasks that re-enter the retain set automatically receive the anchor recorded during their first learning phase, so the metric accommodates arbitrary retain/forget permutations without re-initialisation.

CLU is most useful in the realistic regime where (i) the user may issue multiple, possibly conflicting deletion requests over time, and (ii) storage or privacy policies prevent us from keeping an identical copy of the training data or from re-training a model after every edit. Our experiments in Section 7.5 show that CLU closely tracks the qualitative trends of more expensive baselines while remaining trivial to compute online.

Membership Inference Attacks (MIA)

For MIA testing, we chose to stay consistent with methods in modern literature. We adopted the *Basic MIA* method introduced by (4), adapting it specifically for our

task-incremental unlearning scenario. Our primary objective was to evaluate the effectiveness of unlearning by quantifying residual memorisation through membership inference accuracy.

Given a trained model f subjected to task-based continual learning and subsequent unlearning, the *basic_mia* function trains a binary classifier for each task to distinguish between samples from the forget set $\mathcal{S}$ (labelled 1) and unseen test samples $\mathcal{T}$ (labelled 0). The input to the classifier is the per-sample cross-entropy loss $\ell(x, y)$:

$$\mathcal{D}_{\text{train}}^{\text{binary}} = \{(\ell(x_i, y_i), y_i^b)\}, \quad y_i^b = \begin{cases} 1 & \text{if } x_i \in \mathcal{S} \\ 0 & \text{if } x_i \in \mathcal{T} \end{cases}$$

We construct $\mathcal{S}$ from the specific task being unlearned—i.e., data the model should have forgotten—and draw $\mathcal{T}$ from test samples of the same task to ensure a matched class distribution. Each task is evaluated independently to reflect the sequential nature of task-based learning and unlearning.

The output of the attack evaluation function includes two key metrics:

- Attack Accuracy: The classification accuracy of a binary logistic regression model trained to distinguish between retained and forgotten samples. Higher values indicate that the model can more reliably identify samples in the forget set, implying insufficient unlearning. A baseline of 50% corresponds to random guessing.

- Area Under the ROC Curve (AUC): The AUC provides a threshold-independent measure of attack performance by summarising how well the model ranks members above non-members across all possible decision thresholds. It is computed as the probability that a randomly selected member is assigned a higher score than a randomly selected non-member. AUC is less sensitive to the choice of threshold compared to attack accuracy, offering a more robust assessment of membership leakage.

We used logistic regression as the attack model to maintain consistency with prior work in empirical unlearning evaluation (Appendix D).

7.2 Baseline

To establish a baseline for each comparison we decide to make between unlearning algorithms, we first need to establish a solid foundation of continuous learning algorithms. Our main candidates to act as our continual learning basis were Gradient Episodic Memory (GEM) and Averaged Gradient Episodic Memory (AGEM) before making use of our unlearning algorithms. The main idea behind doing so is to establish the grounds for our new innovative field of continual unlearning.

7.2.1 Continual Learning Initial Exploration

As noted earlier, we wrote our own versions of Naïve Replay, GEM, and AGEM, taking conceptual cues—and minimal code—from Facebook Research's reference implementation at `https://github.com/facebookresearch/GradientEpisodicMemory` (MIT-licensed). The original repo is superb for reproducing published numbers, yet much of its logic is hard-wired to GEM-specific helper classes, so even small changes—let alone grafting a machine-unlearning pipeline—would require invasive surgery. Re-implementing from scratch gives us a lightweight, PyTorch-only codebase whose training loop is deliberately modular: replay buffers, gradient-projection routines, and unlearning blocks can be swapped by changing a single import. It also sidesteps any licensing grey areas and satisfies academic-integrity expectations by demonstrating genuine understanding. Reproduction forced us to create unit tests that match the baseline results, profile bottlenecks (e.g., the QP solver), and shave milliseconds off hot paths, ensuring that when we layer CL $\times$ MU experiments on top, we are working with code we fully control and trust.

To verify that our implementations behaved as intended, we followed the standard continual-learning protocol on CIFAR-10, split into five sequential tasks of two classes each. For each algorithm we first ran a single pass to confirm that the qualitative curves average accuracy and forgetting matched the trends reported in the original GEM and AGEM papers. (Naïve Replay had no canonical paper, and its behaviour is easy to sanity-check by varying the memory budget). After that sanity check, we switched to five independent runs per method. Only once the CIFAR-10 results were consistent did we scale the same code to CIFAR-100, where the larger task-sequence stresses both memory and projection routines. This staged approach

gave us high confidence that any subsequent unlearning experiments would rest on a sound, reproducible baseline.

We started with CIFAR-10 precisely because it keeps turnaround tight. With only five two-class tasks, the GEM QP faces far fewer constraints than it will later, so a full run takes about 40 minutes on our GPU, often less, because we can abort early as soon as verbose log hooks flag a mismatch. That rapid feedback loop lets us edit the codebase, rerun, and re-verify several times a day instead of waiting upward of two hours a single CIFAR-100 run typically needs. Once the small-scale tests were convincingly correct, we locked the code and moved up to CIFAR-100 to confirm complete correctness by acquiring the results produced in their respective academic papers.

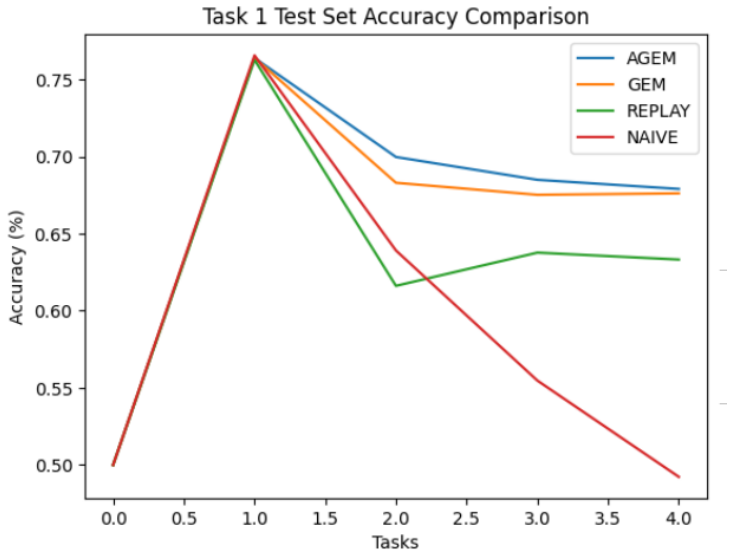


Figure 6: Results from one run on CIFAR-10

Looking at Figure 6, we see that we have successfully implemented naïve replay (REPLAY), GEM and AGEM and compare them to a naïve retraining model. These results are in line with our expectations for the following reasons: We anticipated REPLAY to have worse performance compared to GEM and AGEM due to REPLAY overfitting on the episodic memory it stores, but we expected REPLAY to outperform naïve retraining. Furthermore, we expected to see GEM and AGEM have comparable performance with minimal catastrophic forgetting, with GEM slightly outperforming AGEM. In Figure 6 we can see they have comparable performance and AGEM does slightly better at retaining task 1. We hypothesised

that the reasoning behind this behaivour is due to AGEM getting "lucky" for this particular run. To confirm this, we performed this test, but instead plotted the average accuracy of task 1 over 5 runs.

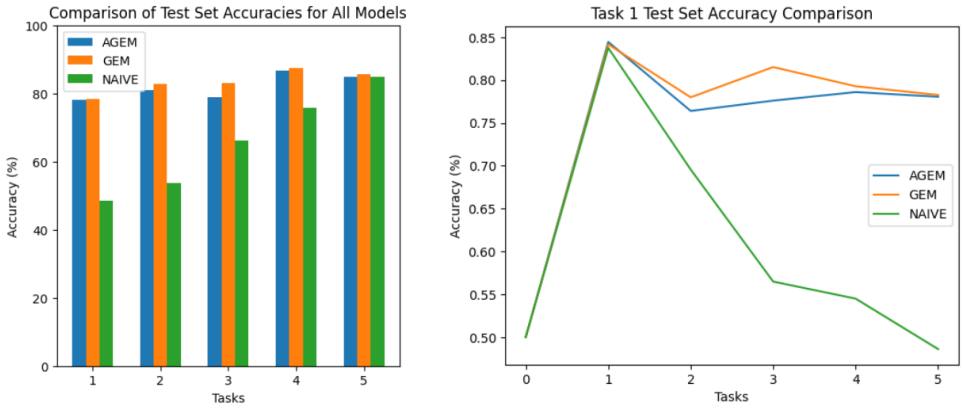


Figure 7: plots of test accuracy of every task and the affect on accuracy of task 1 over iterations using a memory buffer of size 256 for each task

The evidence for this test is presented in Figure 7 and the results support our hypothesis. Another interesting observation is the severity of catastrophic forgetting in naïve retraining models. It justifies the importance and need for continuous learning techniques to be explored and eventually put into practice. When testing our implementation in CIFAR-100 we used the same framework as presented in the GEM paper (17), namely splitting the 100 classes into tasks of size 5, leaving us with 20 tasks that we continually learn on. We plot the results of the average test accuracy of task 1 at each time step over 5 runs.

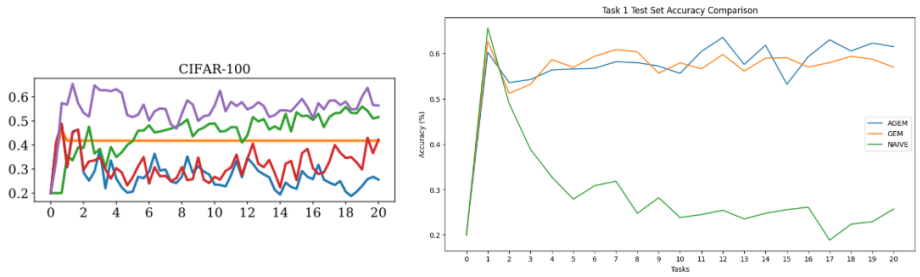


Figure 8: Left: Results Acquired from GEM paper (17). The purple plot is the GEM plot. Right : Our results of AGEM, GEM and naïve retraining respectively

The plot on the left in Figure 8 shows the Lopez-Paz and Ranzato experiments and comparisons of GEM against several other continual learning algorithms.

Namely, Blue [Single] - a single predictor trained across all tasks. Orange [Independent] - one independent predictor per task. Each independent predictor has the same architecture as "single" but with T times less hidden units than "single". Green [iCarl] (28) - is a class-incremental learner that classifies using a nearest-exemplar algorithm. Red [EWC] (8) - where the loss is regularised to avoid catastrophic forgetting. Purple - GEM. The plots on the right demonstrate our results from our implementations using results from an average of five runs. Our implementations of GEM are comparable to the results of GEM presented in the left plot 8, leaving us with confidence that our implementations of GEM are correct. In addition, Chaudhry (15) found that "A-GEM and GEM perform comparably in terms of average accuracy" thus giving us a greater confidence that our implementations of AGEM are also sound.

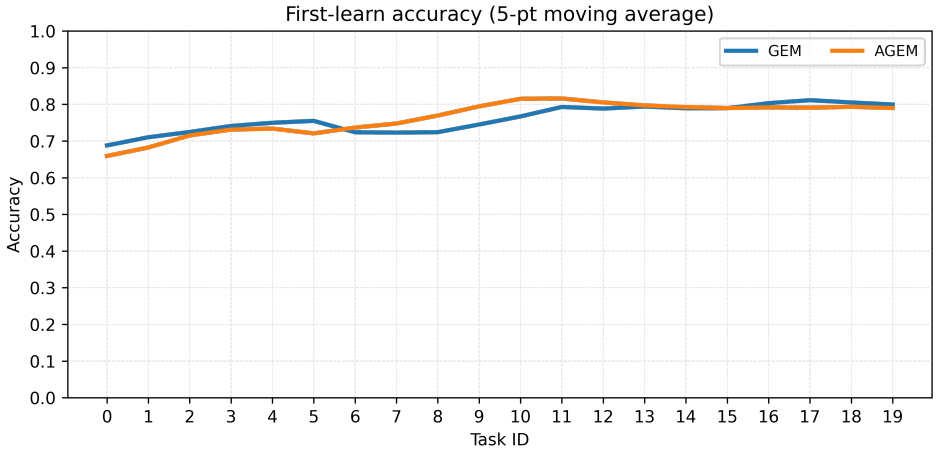


Figure 9: 5-pt moving average of the initial accuracies of tasks over 3 runs

In addition, we wanted to investigate whether there is an influence of learning a task with respect to time. To examine this, we kept track of the moving average of initial accuracies over 3 runs. That is, these are the average accuracies of tasks when they are first learned. Figure 9 shows that there is a positive trend in initial learning accuracies with respect to time, which is supported by the fact that the 5-pt moving average increases. This means that both GEM and AGEM algorithms have significant Forward Transfer (FWT) where learning tasks give incentive to the learning of subsequent tasks. This makes sense as a form of transfer learning is applied to later tasks. In the beginning the model is a randomly allocated weights

whereas when learning later tasks the model has already seen x tasks that all are 5 class image classifiers.

7.2.2 The Need for Continual Unlearning

Why do humans forget? From an evolutionary standpoint the main reasoning behind giving the human mind the ability to forget can be attributed to two reasons:

- Humans do not need to retain all information presented to them, as accessing a single element through a vast selection of a humans collective experience will be largely unnecessary and inefficient.
- At some later point, due to groundbreaking environmental changes humans do not necessarily need to retain old outdated information. This is due to the fact that in reality our environment is always changing meaning what defines "useful" and "useless" information is non-static.

We claim that a truly efficient continual learning model must be as effective at learning new tasks as it is at forgetting old ones. A model that can efficiently forget old tasks would be able to make room within their episodic memory to learn tasks. Furthermore, one of the main downsides of the GEM algorithm was the long continual learning time as the number of memories within the memory buffer increased for every learnt task. However, with the power to be able to perform targetted forgetting of old outdated tasks we could alleviate this problem. Additionally, forgetting old tasks will allow to to use the negatives of the phenomenon known as "catastrophic forgetting" as one of our strengths as opposed to a weakness of continual learning.

7.2.3 Formalisation of the Continual Unlearning Problem

We define the continual unlearning problem as follows, where we have a continual learner that may have learned a set of tasks $T = \{T_1, T_2, \ldots, T_n\}$ in a sequential manner. Then once that set of tasks has been learned, we may realise that we wish to unlearn a subset of tasks $T_{unlearn} = \{T_{unlearn_1}, T_{unlearn_2}, \ldots, T_{unlearn_k}\}$. We define the set of tasks that we want to keep as $T_{retain} = T \setminus T_{unlearn}$.

Our goal is to learn a model f that can learn the tasks in T and then unlearn the tasks in $T_{unlearn}$ while retaining its utility on the tasks in T_{retain}. The main

evaluation metrics for continual learning will depend on what we seek to gain from each un-learned model.

- **Overall Model Utility:** This is the overall model utility after unlearning given by our Continual Unlearning Tug-of-War (CL-ToW) metric defined previously. The ToW metric combines the performance on retain, forget, and test set in a single metric.
- **Imperviousness to Membership Inference Attacks:** This is the ability of the model to resist membership inference attacks on tasks in $T_{unlearn}$.

In addition to the unlearning phase, we also need to take into account that performing continual learning and continual unlearning can occur in any ordering. Our tests should reflect this fact to take into account the fact that a task can be forgotten at any point in time. Therefore, we must define our testing to have results that specify a diverse range of task sequences and scenarios.

> **i.e.** by *unlearning* what a model has learnt about a task, we can *relearn* what the model has forgotten about the previous tasks. Hence the possibility of 'Relearning by Unlearning' (RBU).

In fact, GEM managed to achieve positive backward transfer in very specific scenarios where the sequence and ordering of tasks and classes within those tasks nurtured the performance and utility on previous tasks (6).

7.3 Unlearning-GEM for GEM & Unlearning-AGEM for AGEM

We provide two main variants of the continual unlearning algorithm pipelines, most notably the NegGEM and NegAGEM algorithms. The AGEM with NegAGEM pipeline exists as the efficient variant of the GEM with NegGEM pipeline. We justify this combination of algorithms with the fact that one is based off of an averaging gradient method which is more efficient but more sensitive to sudden changes in the gradient direction. The other is based off of gradient episodic memory with the solving of a constraint minimisation problem which is more robust to sudden changes in the gradient direction but takes longer to compute as it has to solve a CPU-bound quadratic program. Hence, it is only a natural choice to combine the two algorithms to form hybrid algorithm pairs.

However, based on the implementations of the two algorithms they have differing hyperparameters and settings. For example, the AGEM algorithm had an original hyperparameter of 1300 total memories used as the total buffer size leaving 65 memories per task within the CIFAR-100 dataset (15).

In contrast to this, the GEM algorithm had a total episodic memory size of 5120 memories with 256 memories per task (6). This is a significant difference in the number of memories used for each algorithm and therefore we need to take this into account when performing our continual unlearning experiments. Our expectation is that whilst retain performance may be similar due to the difference in continual learning and unlearning buffers sizes, the forgetting performance will be more optimal using the NegGEM algorithm simply due to the fact that we have more forgetting samples to work with. On the other hand, we mainly want to observe if an efficient continual unlearning alternative is feasible.

7.4 Experimental Setup

To test the performance of our continual unlearning algorithms we will be using the CIFAR-100 dataset with a continual learning and unlearning task sequence of 20 tasks.

Task Sequence What is a task? Quite simply, in task incremental learning, a task is a subset of classes from the CIFAR-100 dataset which we can learn and unlearn. For example the first task could be the classes 'apple', 'orange', 'banana' and 'grape' and the second task could be the classes 'car', 'truck', 'bus' and 'motorbike'. For our case, the tasks themselves will be filled with a randomised selection of 5 classes from the CIFAR-100 dataset, with each class only appearing once. The task sequence is defined as the order in which we learn and unlearn tasks. For example, the task sequence could be '0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -9, -8, -7, -6, -5, -4, -3, -2, -1'. This is equivalent to learning tasks 0-9 and then unlearning tasks 9-1 in sequential order.

The Original Continual Learning Sequence The goal of the continual learning is to ensure a model learns a sequence of tasks without catastrophic forgetting. The reason as to why Lopez-Paz (6) chose to use a dataset with such a large number of classes was to ensure that even after learning a large 'continuum' of data the

utility and performance of the model on the first task was still retained. This is a very important aspect of continual learning as it ensures that the model is able to learn a large number of tasks without forgetting earlier tasks, with the first task being the worst case.

We need to be able to extend on this idea to prove that we can both learn *and* unlearn a continuum of data without forgetting earlier tasks. The best case scenario is that by unlearning a continuum of data that we end up with positive backward transfer on the first task. We predict this to be extremely unlikely as we are effectively doubling the extent of the task sequence. Additionally, whilst in (6) alongside the memory buffers they were also able to 'observe' the 2500 samples per task at least once, we aim to be able to unlearn the task using only the memory buffers and selectively choosing which memorised samples to unlearn. This is a very important aspect of our continual unlearning algorithms as it allows us to more effectively unlearn a task with fewer samples (21).

> **i.e.** we can unlearn a task with only the memory buffers and a small number of samples from the original task.

7.4.1 Tuning Hyperparameters for GEM & AGEM vs Unlearning-GEM & Unlearning-AGEM

In general, finetuning hyperparameters for continual learning algorithms was not as arduous as we had originally thought. Quite simply, we based our hyperparameters off of the original GEM and AGEM algorithms (6) and (15) respectively. On the other hand, when fine tuning the hyperparameters for a continual unlearning equivalent turned out to be much more difficult based empirically on the sensitivity of tuning NegGrad+ [1] from (21). Changing hyperparameters such as the learning rate, optimiser, and batch size can have a significant impact on the performance of the continual unlearning algorithm. It effectively became a game of trial and error to find the best hyperparameters for the continual unlearning algorithms. What it ultimately came down to was balancing continual unlearning performance whilst retaining performance on tasks within our retain set. In addition to this, the combination of a Saliency Unlearning threshold γ mask alongside the buffer composition

[1] From here on out we interchangabley use GEM+ and NegGrad+. Whilst NegGrad+ is actually an unlearning algorithm, we use this name in the context of GEM+

of our continual learning and unlearning buffers was also another important set of hyperparameters to tune dependent on the variation of the algorithm (5).

7.4.2 Selecting the Saliency Unlearning Threshold γ

Saliency Unlearning is a method that allows us to unlearn a task by removing the most salient values from the gradients of the task we are unlearning. We opted to use the hard thresholding method as it is the most efficient however, soft thresholding could have also been considered in cases where we wanted to retain some of the salient values or where 0 values in our gradients were not ideal.

7.4.3 Impact of Saliency–Based Gradient Sparsification on GEM Constraint Violations

We additionally probed the idea that utilising SalUn could in fact be used to reduce the total number of GEM constraint violations. This was based on the given knowledge that a forgetting gradient vector that is majority 0 may increase the number of unlearn steps required, as the overall gradient magnitude is smaller, but would decrease the number of violations per iteration. The idea is based on the fact that a GEM constraint problem with a majority 0 values in the forget vector would be easier to solve as the norm of the Euclidean distance would be smaller assuming majority opposing gradients between a forgetting gradient and a memorisation gradient.

GEM constraint recap. Given an episodic memory containing per-task gradients $\{g_k\}_{k=1}^{K}$, Gradient Episodic Memory (GEM) updates a model by projecting the raw gradient g of the incoming task onto the feasible cone

$$\mathcal{C} \;=\; \big\{\, v \in \mathbb{R}^p \mid v^\top g_k \;\geq\; 0,\; \forall k \in \{1,\dots,K\} \big\}.$$

A *constraint violation* occurs when $g \notin \mathcal{C}$, equivalently when $g^\top g_k < 0$ for at least one stored task k.

SalUn masking. Saliency Unlearning (SalUn) replaces g by a sparsified gradient g', obtained through hard thresholding with parameter $\gamma > 0$:

$$g'_i \;=\; \begin{cases} g_i, & |g_i| \geq \gamma, \\ 0, & |g_i| < \gamma, \end{cases} \qquad g' \;=\; M(\gamma)\, g,$$

where $M(\gamma) = \mathrm{diag}(m_1, \ldots, m_p)$ has $m_i \in \{0, 1\}$ and sparsity[2] $1 - \rho$.

Dot–product statistics. Define the random variables

$$X \;=\; g^\top g_k, \quad X' \;=\; g'^\top g_k.$$

Assuming weakly dependent coordinates with finite second moments, the multivariate central-limit theorem yields

$$X \,\dot{\sim}\, \mathcal{N}\big(\mu,\, \sigma^2\big), \qquad X' \,\dot{\sim}\, \mathcal{N}\big(\mu',\, \sigma'^2\big),$$

with

$$\mu' = \mu - \sum_{i:\, m_i = 0} \mathrm{E}[g_i g_{k,i}], \quad \sigma'^2 = \sigma^2 - \sum_{i:\, m_i = 0} \mathrm{Var}(g_i g_{k,i}).$$

Because the masked coordinates are chosen purely by magnitude, they have negligible influence on the mean but non-trivial influence on the variance; consequently $\mu' \approx \mu$ and $0 \leq \sigma'^2 \leq \sigma^2$.

Violation probability. For $Z \sim \mathcal{N}(\mu, \sigma^2)$ we have $\Pr[Z < 0] = \Phi(-\mu/\sigma)$, where Φ is the standard normal cdf. Hence

$$\Pr[X' < 0] \;=\; \Phi\Big(-\tfrac{\mu'}{\sigma'}\Big) \;\leq\; \Phi\Big(-\tfrac{\mu}{\sigma}\Big) \;=\; \Pr[X < 0],$$

since $\sigma' \leq \sigma$ and $\mu' \approx \mu$.

Proposition. *Provided masking does not materially change the sign of the mean inner-product, saliency-based sparsification weakly* reduces *the likelihood of a GEM*

[2] We denote the *retained* coordinate fraction by $\rho = \frac{1}{p} \sum_{i=1}^{p} m_i$.

constraint violation:

$$\Pr[X' < 0] \;\leq\; \Pr[X < 0].$$

Pathological corner case. The inequality can reverse if the retained coordinates are systematically *anti-aligned* with the stored gradients, shifting μ' left while also decreasing σ'. This can occur when (i) large-magnitude components of the new task gradient oppose many past tasks, and (ii) the threshold γ is alignment-agnostic.

Practical implication. Increasing γ (hence lowering ρ) therefore *tends* to cut both projection frequency and correction magnitude in GEM, accelerating training while preserving feasibility. Nonetheless, monitoring the empirical violation rate remains essential; if violations rise, one can lower γ or employ a *signed* mask that preserves salient coordinates whose directions are consistent with historical gradients.

7.4.4 Task Sequence Testing Scenarios

Why We Execute the Full Battery of Task Sequences

Machine–unlearning algorithms face *orthogonal stressors* that seldom co-occur in standard benchmarks. The eight sequences (A–H) constitute a **minimal yet covering test-suite** that spans the full design space along three principled axes:

1. **Stress testing extreme volatility (A, B).** Sequence A deletes a *contiguous* block of 19 tasks in reverse order, while Sequence B alternates *learn→forget→relearn* for each task. Together, they create the most demanding retain–forget tug-of-war and expose an algorithm's limits under heavy gradient conflict and rapid state changes.

2. **Mid-stream splicing (C, D).** Both curricula cut a "slice" out of the task timeline and then continue training: C removes a central chunk (tasks 10–14) late in the stream, whereas D removes an earlier chunk (tasks 5–9) soon after it is learned. The pair therefore isolates how the *position* of a deletion within the lifetime of the model influences subsequent plasticity.

3. **Residual-knowledge effects (F, G).** Both sequences deliberately *re-insert* tasks that have just been forgotten: F deletes tasks 5–9 and relearns them

after a block of new tasks, while G deletes the same tasks and relearns them *immediately*. Comparing the two reveals whether any hidden trace of the discarded weights accelerates or hinders future re-learning.

4. **Temporal-position sensitivity (H).** Sequence H erases four periodic outliers (tasks 0, 4, 9, 14) long after they were learned. Because the forgotten tasks are spaced evenly through the chronology, H tests whether "age" alone (time-since-learning) makes certain memories harder to expunge or retain.

Running the full suite therefore yields a *multi-facet score* covering the three core desiderata of modern unlearning research—**forget quality**, **retain utility**, and **computational efficiency**—under every practically relevant corner-case. Any method that performs robustly across A–H can be trusted to handle real-world deletion requests whose structure is *a priori* unknown.

Task Sequence A

$$\begin{aligned}\text{Task Sequence} = \{&0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,\\ &-19, -18, -17, -16, -15, -14, -13, -12, -11, -10,\\ &-9, -8, -7, -6, -5, -4, -3, -2, -1\}\end{aligned} \tag{7.3}$$

Scenario & Need. The model first acquires a contiguous curriculum of 20 tasks ($0 \to 19$) and is then asked to *fully forget* tasks 1–19 in the *reverse order*. This mimics a "right-to-be-forgotten" request that arrives long after deployment, where only the founding capability (task 0) must be retained. Because the forget set is (i) large, (ii) contiguous, and (iii) heavily *entangled* with the retain set in representation space, unlearning difficulty is expected to be maximal (see the entanglement analysis of (21)).

Task Sequence B

$$\begin{aligned}\text{Task Sequence} = \{&0, 1, -1, 1, 2, -2, 2, 3, -3, 3, 4, -4, 4, 5, -5, 5, 6, -6, 6, 7, -7,\\ &7, 8, -8, 8, 9, -9, 9, 10, -10, 10, 11, -11, 11, 12, -12, 12, 13, -13, 13,\\ &14, -14, 14, 15, -15, 15, 16, -16, 16, 17, -17, 17, 18, -18, 18, 19\}\end{aligned} \tag{7.4}$$

Scenario & Need. The stream alternates between *learn* and *unlearn* of the *same* task, e.g. $\{1, -1, 1, 2, -2, 2, \dots\}$. This emulates a volatile production system where customers repeatedly revise their data-retention preferences. The algorithm must switch state *quickly*, with strict limits on memory footprint and number of gradient steps.

Task Sequence C

$$\begin{aligned}\text{Task Sequence} = \{&0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 12,\ 13,\ 14,\\ &-14,\ -14,\ -13,\ -12,\ -11,\ -10,\ 15,\ 16,\ 17,\ 18,\ 19\}\end{aligned} \tag{7.5}$$

Scenario & Need. After learning tasks $0-14$, the system discovers that task 14 and its immediate neighbours (10–13) are poisoned. They are unlearned mid-stream before new tasks $15-19$ arrive. This tests whether an algorithm can *forget deep-in-the-past, highly-memorised, topologically central data* and then *regain plasticity* for future learning. Memorisation level is high because the bad data lived many epochs in the network (21).

Task Sequence D

$$\begin{aligned}\text{Task Sequence} = \{&0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\\ &-9,\ -8,\ -7,\ -6,\ -5,\ 10,\ 11,\ 12,\ 13,\ 14,\ 15,\ 16,\ 17,\ 18,\ 19\}\end{aligned} \tag{7.6}$$

Scenario & Need. A GDPR request arrives *early*: after task 9 the user wants tasks 5–9 removed. Training must continue to task 19 *without future re-initialisation.* Hence forward transfer and spare capacity are crucial.

Task Sequence E

$$\begin{aligned}&\text{Random Task} = \mathit{random.randint}(0, 19)\\ &\text{Task Sequence} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, -\text{Random Task}\}\end{aligned} \tag{7.7}$$

Scenario & Need. Single random task deletion after the full curriculum — the classical benchmark adopted by many approximate unlearning papers.

Task Sequence F

$$\begin{aligned}\text{Task Sequence} = \{&0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ -9,\ -8,\ -7,\ -6,\ -5,\\ &10,\ 11,\ 12,\ 13,\ 14,\ 15,\ 16,\ 17,\ 18,\ 19,\ 5,\ 6,\ 7,\ 8,\ 9\}\end{aligned} \tag{7.8}$$

Scenario & Need. After forgetting tasks $9-5$ the user re-evaluates and decides the data were *lawful*. The algorithm must *re-insert* the same tasks at the end of training. This probes *reversibility* and cumulative cost.

Task Sequence G

$$\begin{aligned}\text{Task Sequence} = \{&0,1,2,3,4,5,6,7,8,9,-9,-8,-7,-6,-5,10,11,12,13,14,5,6,7,\\ &8,9,15,16,17,18,19\}\end{aligned} \tag{7.9}$$

Scenario & Need. Partial unlearning of tasks $9-5$ is followed *immediately* by their re-introduction *before* new tasks 15–19 arrive. This tests whether an algorithm can juggle unlearning and rapid forward transfer simultaneously.

Task Sequence H

$$\begin{aligned}\text{Task Sequence} = \{&0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 12,\ 13,\ 14,\ 15,\ 16,\ 17,\ 18,\ 19,\\ &-0,\ -4,\ -9,\ -14\}\end{aligned} \tag{7.10}$$

Scenario & Need. Deletion targets are sparse and *periodic* (tasks 0, 4, 9, 14), each being a likely *memorised outlier* according to memorisation theory (21). The main challenge is to erase them without damaging intervening tasks that share very few parameters.

7.5 Results

In this section we will be discussing the results of our continual unlearning algorithms on each of the task sequences. For hyperparameter tuning we will be using the task sequence A as our base case.

The specific set of hyperparameters was outputted from the grid search we performed on the task sequence A. The set of hyperparameters used for all sequences and algorithms is outputted for every run of the code as a separate file denoted by the regex '*HYPERPARAMETERS.csv'. However, in the vast majority of cases, hyperparameters that were optimal for the task sequence A were also optimal for the other task sequences.

We provide the optimal hyperparameters used for each algorithm in table 2.

(a) Hyper-parameters identical for all algorithms

Parameter	**Value**
Optimiser	SGD
Learn Epochs	3
Learn Buffer Comp.	0.2
Learn Buffer Selection	Least
Early-stop Threshold	0.7
Learn Batch Size	10
Unlearn Epochs	24
Unlearn Buffer Comp.	1.0
Unlearn Buffer Selection	Most
SalUn Threshold	0.02

(b) Values that differ between algorithms

Parameter	NegGEM	NegAGEM	RL-GEM	RL-AGEM	ALT-NegGEM	NegGrad$^+$
CL Algorithm	GEM	AGEM	GEM	AGEM	GEM	GEM
Learning-rate	0.10	0.03	0.10	0.03	0.10	0.10
CL Buffer Size	5120	1300	5120	1300	5120	5120
Mem./Task	256	65	256	65	256	256
Unlearn LR	0.01	0.003	0.01	0.003	0.01	0.01
Unlearn Batch	2	2	2	2	2	4
Alpha	N/A	N/A	N/A	N/A	N/A	0.5/0.95

Table 2: Compact presentation of the hyper-parameters. Table (a) lists settings shared by every method; table (b) shows only the parameters whose values differ across algorithms. These are only optimal hyperparameters for test sequence A. For a specific test check the '*HYPERPARAMETERS.csv' for each test.

7.5.1 Long Term Results of Unlearning (Stress Testing)

In order to calculate the values for CL-TOW we need to first calculate the values for the average test and train accuracy of each task over our average of 3 runs.

What was interesting to note was that the average test accuracy of task 1 stayed relatively stable over the course of the unlearning process, except for the final few iterations where it dropped significantly. This is likely due to the fact that eventually the model will start overfitting to the task 1 memory buffers after utilising them for a long period of time. This is a common problem with continual learning algorithms that use some form of episodic memory.

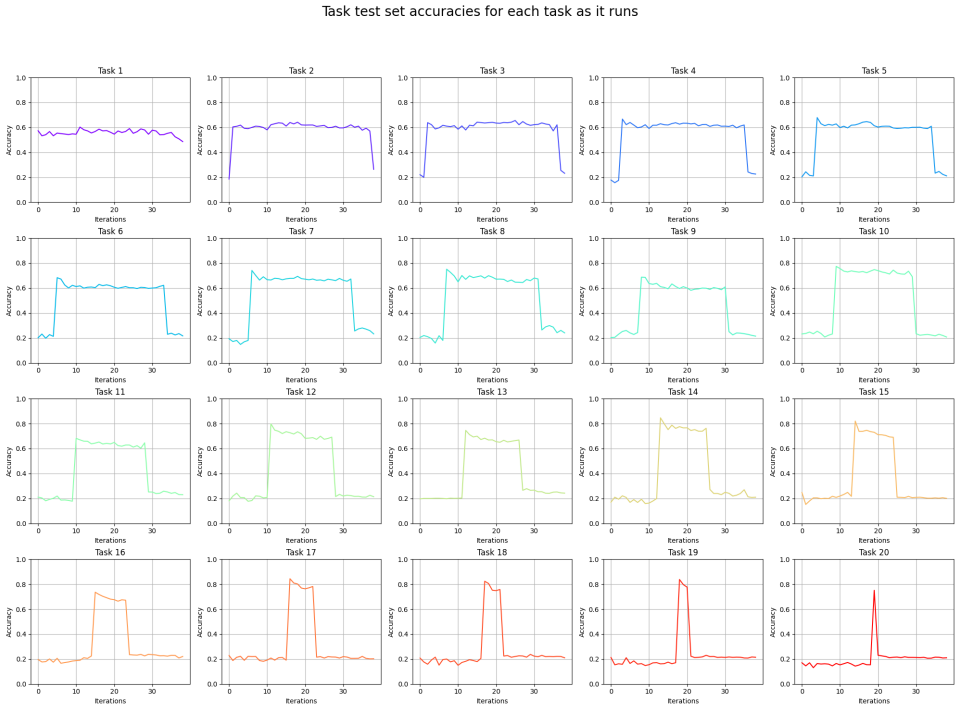


Figure 10: We track the average test set accuracy over all tasks over 3 runs using NegGEM, important to note that after the continual learning phase (iteration 0-20), when unlearning the model is able to retain the accuracy of all tasks within the retain set.

We also provide the average of forget and retain accuracy over the course of the unlearning process to make the above plot easier to interpret the performance of NegGEM. The results are shown in Figure 11.

Continual Learning ToW score as defined in our previous section is calculated based on average absolute differences between a continual learner that has learnt up to that task and a continual learner that has learnt all tasks and then selectively forgotten up to that point. i.e. as an example, a continual learner that only learns tasks 1-5 is equivalent to a continual learner that learns tasks 1-20 and then unlearns tasks 20-6 as their forget and retain sets would be identical. We see that NegGEM

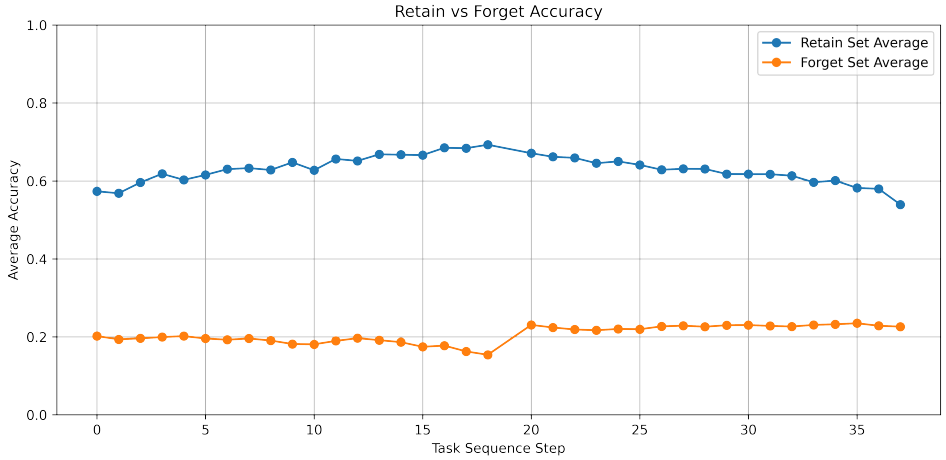


Figure 11: The average over the evolving retain and forget sets for ease of interpretation Using NegGEM continually learning 20 tasks of CIFAR-100. It may appear as though our model is overfitting to the task 1 memory buffer, but the reality is that the model is reverting to the original state where it was trained on task 1 on iteration 1.

was consistently able to achieve CL-TOW scores of above 0.85 during almost the entire unlearning process. We achieve this by using a combination of strict saliency unlearning and a memorisation aware unlearning buffer alongside our continual learning buffer containing 10% of the original train data each respectively. This was applied on all algorithms. We also note the CL-TOW scores for Subset NegGrad+ with alpha values of 0.5 and 0.95 respectively as baseline scores.

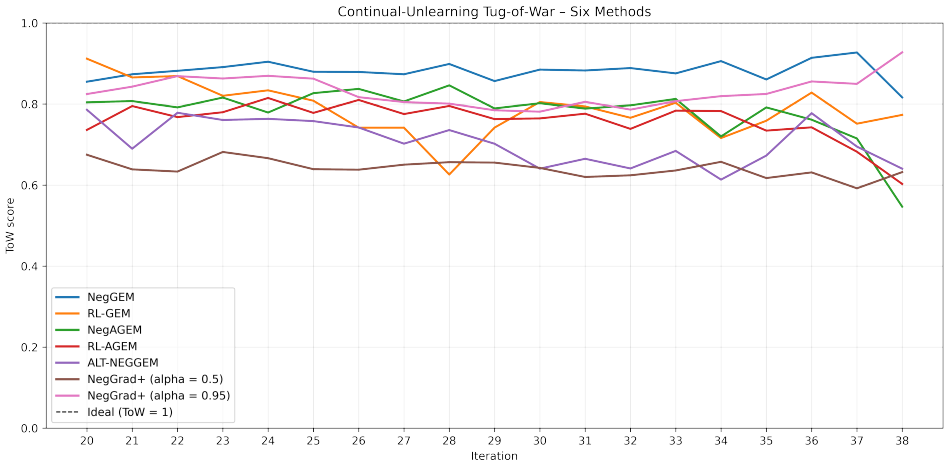


Figure 12: The CL-TOW scores for the NegGEM algorithm. We see that the CL-TOW scores are consistently above 0.85 for the entire unlearning process.

The CL-TOW scores for the Subset NegGrad+ algorithm are shown in Figure

12. We observed that the NegGrad+ algorithm performed significantly worse in terms of CL-TOW scores when the hyperparameter alpha was set to 0.5. This was mainly as a result of the model having poor model retention when the magnitude of the memory gradient was small compared to the negated forget gradient. Whilst the model was able to achieve strong forgetting metrics on its forget set, its retain accuracy was significantly lower than that of the other algorithms.

For the Subset NegGrad+ algorithm with alpha set to 0.95, we see that the model was able to achieve extremely high CL-TOW scores only being second to NegGEM combined with a small weight random labeling step and negated gradient RL-GEM. However, with the alpha value set to 0.95, with the batch size being 8 per iteration, the magnitude of the forgetting gradient in proportion to the magnitude of the retain gradient was extremely small. Although yes, the model achieved optimal retention accuracy whilst being able to unlearn effectively, the compute cost of the algorithm was extremely high due to the small step size of the forgetting gradient.

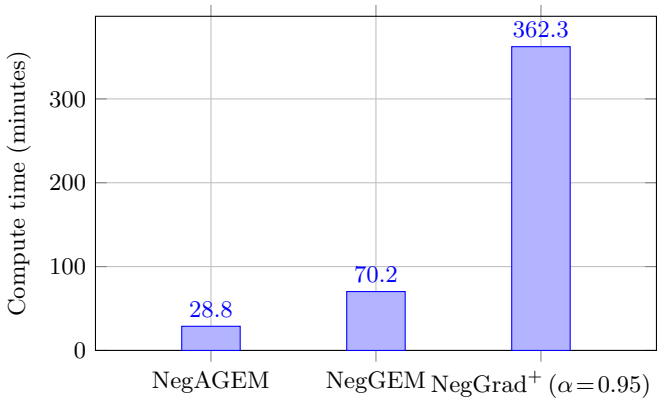


Figure 13: Wall-clock time to complete task sequence A. For clarity, RL-AGEM will have identical compute times to NegAGEM. Alt-NegGEM and RL-GEM have identical compute times to NegGEM.

On the other hand, increasing the size of the relative step by increasing learning rate would massively increase the rate of instability of the model and lead to catastrophic forgetting of the retain set. This is similar to the results of what we observe when using the Subset NegGrad+ algorithm with alpha set to 0.5. Therefore, whilst the Subset NegGrad+ algorithm with alpha set to 0.95 was able to achieve strong CL-TOW scores, it was at the cost of a significant increase in compute time to resolve the model and yet it still performed worse than the NegGEM algorithm except for retention on the last task. In the general case however, there is more to

an algorithms performance than simply one task sequence. In actuality for certain later tests the 0.95 alpha value became extremely restrictive for weight updates effectively meaning that the model failed to unlearn completely. This is where setting the alpha value to 0.5 was more performant across the wide range of various tasks within the test suite. This is why we opted to utilise it for later task sequences.

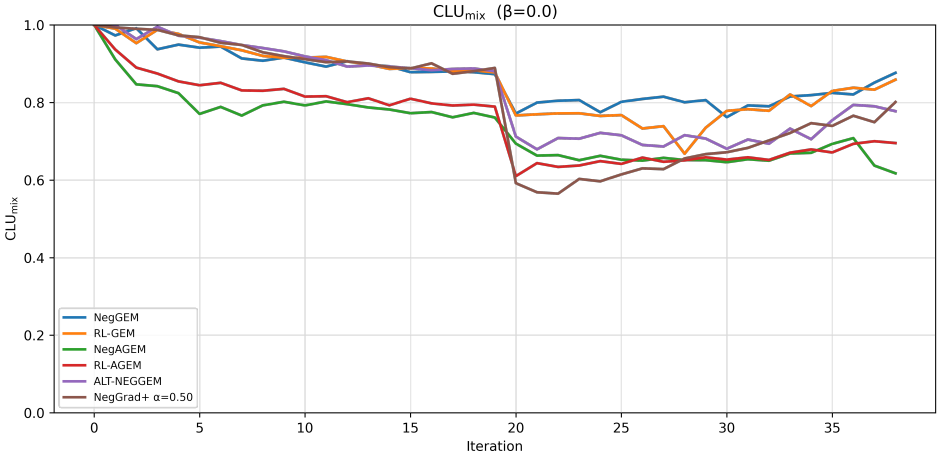


Figure 14: $\mathrm{CLU}_{\mathrm{mix}}$ scores for task sequence A: Comparison of unlearning algorithms over the complete learn-unlearn curriculum.

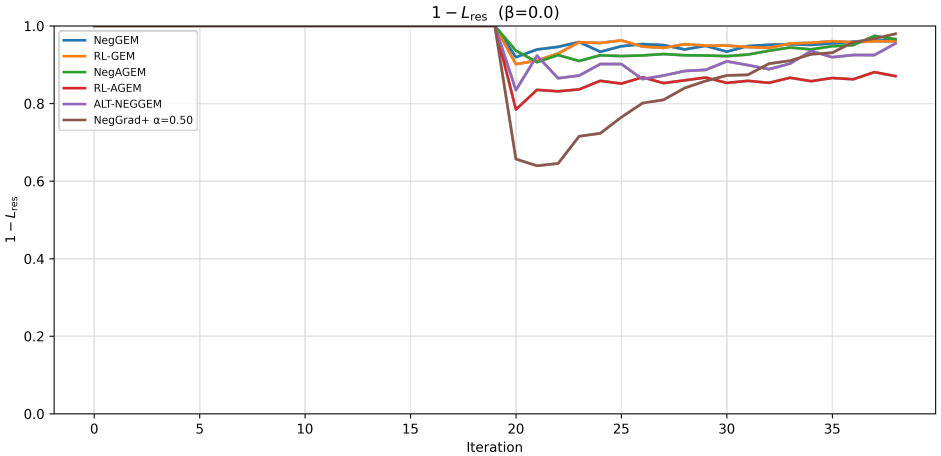


Figure 15: $1 - L_{\mathrm{res}}$ scores for task sequence A: Forgetting effectiveness of each algorithm on the forget set.

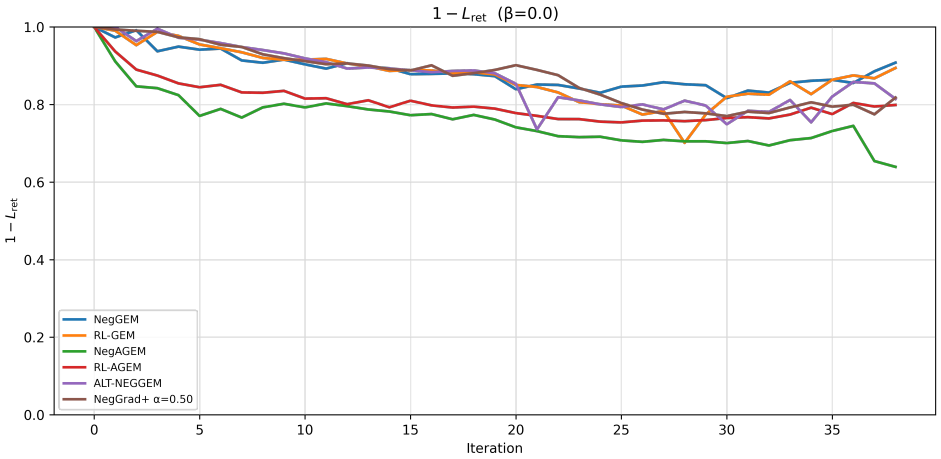


Figure 16: $1 - L_{\text{ret}}$ scores for task sequence A: Knowledge retention capability of each algorithm on the retain set.

Figures 14, 15 and 16 demonstrate that across the A curriculum all methods drive the forgotten–set accuracy down to the random-guess floor after roughly eight unlearning iterations, so the trajectories mainly diverge in how well they preserve the survivor tasks. NegGEM and RL-GEM exhibit the flattest retention curves, dipping to ≈ 0.78 immediately after the first deletion step and then climbing back above 0.80; their composite $\text{CLU}_{\text{mix}}(t)$ therefore remains the highest throughout. AGEM-based variants track the same forgetting profile but lose an additional $\sim$10 pp of retention, while the naïve NegGrad^{+} baseline shows the steepest decline (to $<$0.65) and never recovers. ALT-NegGEM also follows the GEM family until iteration 28. Apart from this isolated event all curves are remarkably smooth, confirming that the memory-buffer replay keeps stochastic variance low even under the large, contiguous forget set.

7.5.2 Forgetting Right After Learning (Stress Testing)

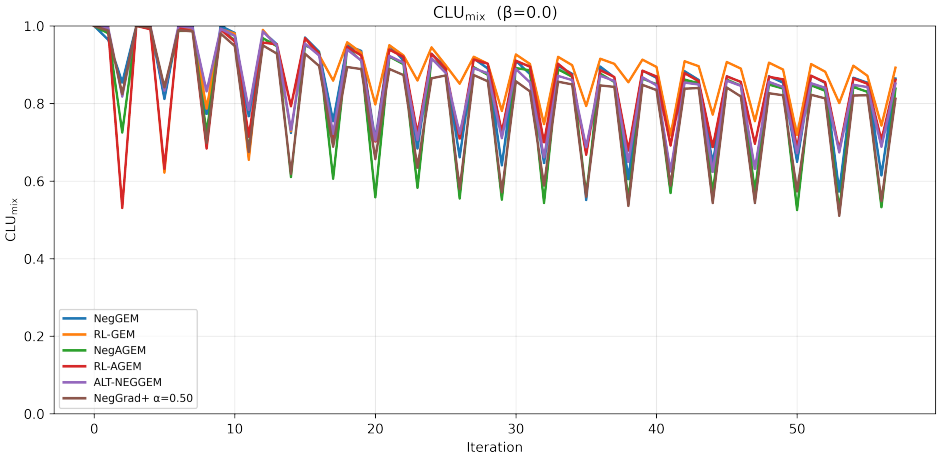


Figure 17: $\mathrm{CLU}_{\mathrm{mix}}$ scores for task sequence B: Algorithm performance during alternating learn-unlearn cycles.

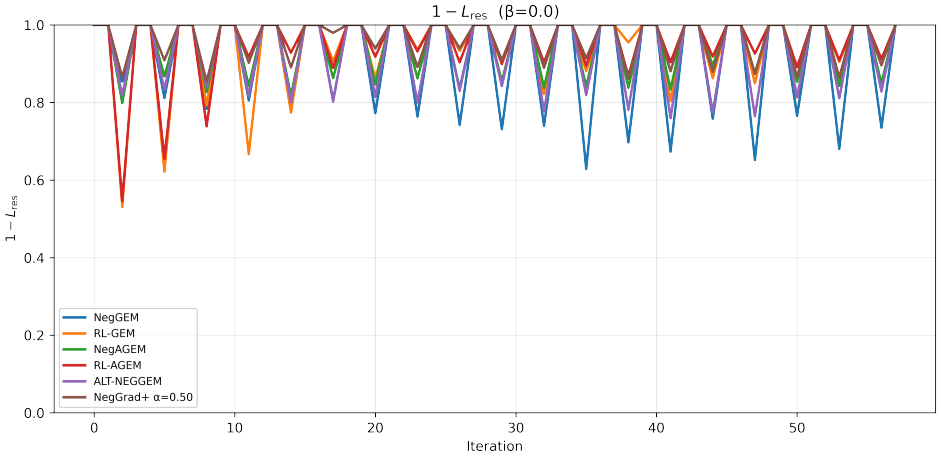


Figure 18: $1-L_{\mathrm{res}}$ scores for task sequence B: Forgetting effectiveness during volatile task transitions.

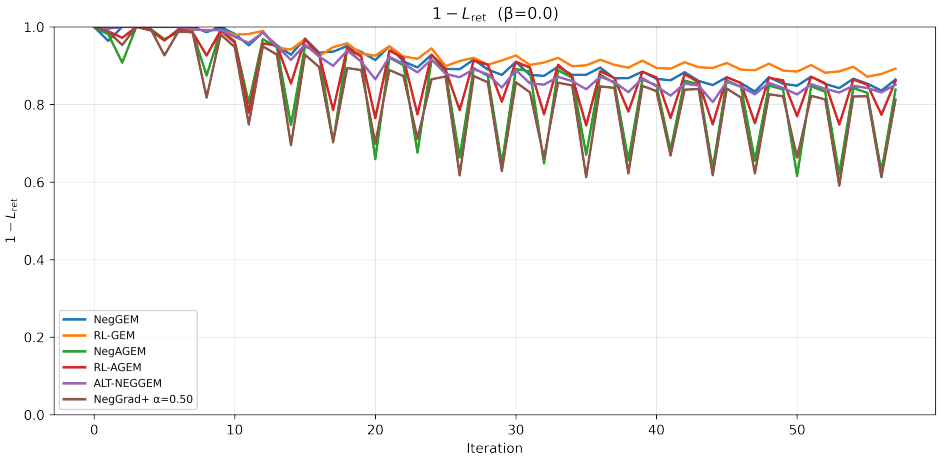


Figure 19: $1 - L_{\text{ret}}$ scores for task sequence B: Knowledge retention stability during frequent preference reversals.

Sequence B was designed to be our longest and most compute intensive setting: every time a continual learner observes a task it immediately forgets it followed by another re-observation of the same task. Thus we produce a sequence of 20 rapid *learn–unlearn* reversals on CIFAR-100. [3]

Figures 17–19 track the resulting CLU_{mix}, $1 - L_{\text{res}}$ and $1 - L_{\text{ret}}$ scores.

All methods drive the forgotten-set accuracy to chance level within one or two deletion steps, so their trajectories mainly diverge in *how abruptly* they lose and then re-acquire survivor knowledge.

Per Algorithm Performance Analysis In this case, RL-GEM features both the flattest retention scores alongside some of the most pertinent forgetting performance amongst the GEM variants. The initial dip never falls below ∼0.74 and the line quickly climbs up back up above 0.80 after every second re-learning phase , yielding the top CLU_{mix} envelope throughout (orange line in Fig. 17). We suspect RL-GEM to be the most performant on this specific test, due to the fact that its retention on previous tasks is significantly stronger compared to NegGEM. In general, random labeling has better retention performance using a subset of the data because a random gradient is significantly less likely to have a constraint violation compared to a negated gradient. RL-GEM does not aim to maximise losses but instead swaps the labels around to ensure that the model does not know which

[3] As a result of this, the curriculum is therefore highly volatile: at iteration t the survivor set of all tasks contains $\{1, \ldots, t\}$.

class to predict. It maximises confusion as a result of training it to predict one classifier at one iteration and then trains it to predict another classifier in the next. We found that the random labelling technique is most effective at forgetting after recently observing a task except for iteration 2 where RL-GEM fails to unlearn. It appears as though RL-GEM is most effective at formulating model confusion when the number of observed tasks is large but the worst in the first few iterations.

NegGEM exhibits the exact opposite behaviour, with strong forgetting performance on the earlier tasks which gradually worsens with a growing number of tasks. This may be as a result of the fact that as the number of tasks increases, the number of GEM constraints that must hold with regards to the negative gradient also increases. Thus, the unlearning performance worsens.

AGEM variants(green and red) follow the same $1 - L_{\text{res}}$ trajectory but shed an extra $\sim$12 pp of retention immediately after each deletion, which is never fully recovered before the next wipe-out step begins. Consequently their CLU_{mix} values oscillate between 0.60 and 0.70. In general, AGEM variants are more sensitive to constant task-switching and rapid *learn–unlearn* cycles. Alt-NegGEM initially follows the trend of NegGEM but then also follows the same problem encountered by the AGEM variants.

7.5.3 Forgetting In-between Learning (C)

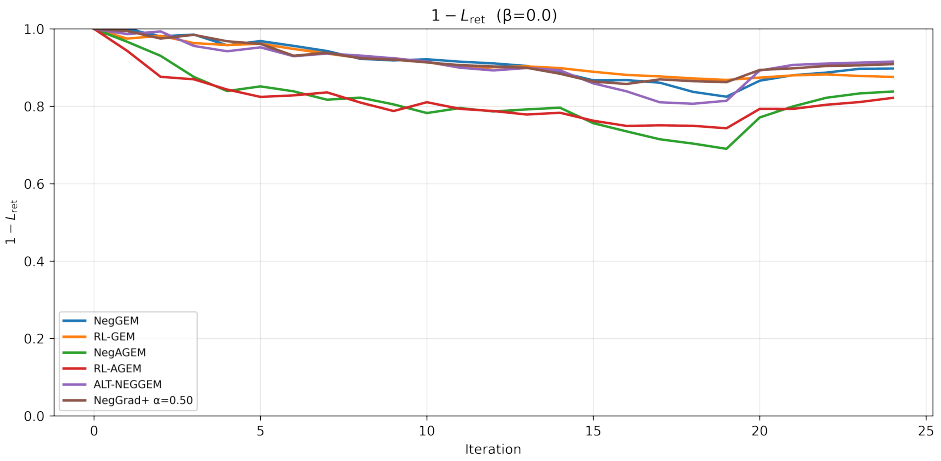


Figure 20: $1 - L_{\text{ret}}$ scores for task sequence C: Retention capability after unlearning central tasks.

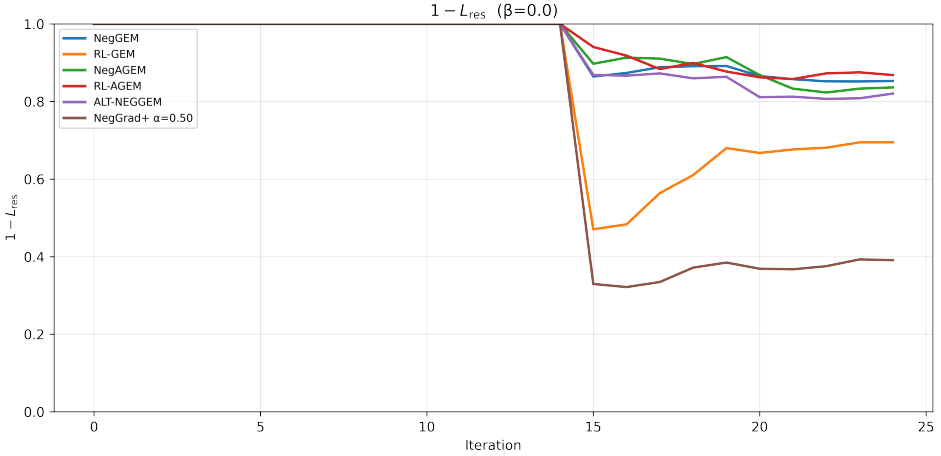


Figure 21: $1 - L_{\text{res}}$ scores for task sequence C: Retention capability after unlearning central tasks.

The C setup was introduced to isolate the impact of performing unlearning mid-training on subsequent learning. To assess this, we compare the final test accuracies of the first 10 tasks and the last 5 tasks between the C setup and the first 20 iterations of the A setup, where all 20 tasks are learned sequentially without interruption. This comparison allows us to determine whether unlearning during training has a tangible effect on the retention of tasks learned before unlearning and on the acquisition of tasks learned afterward.

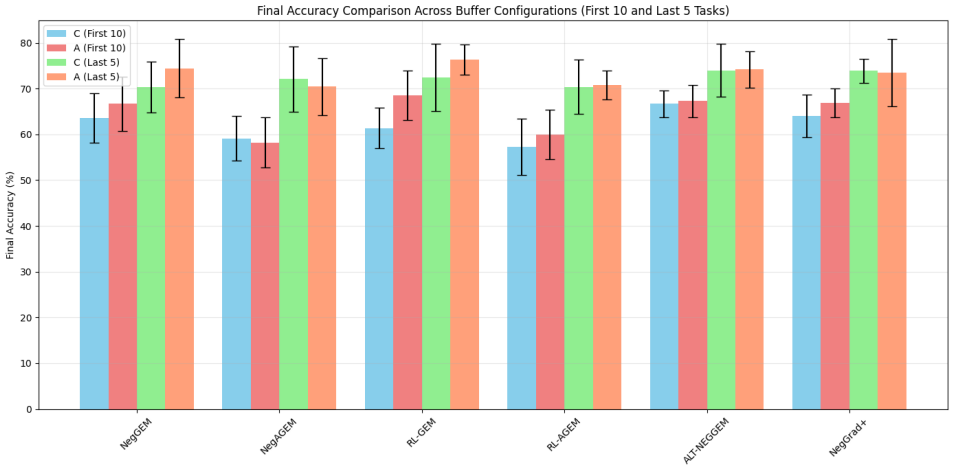


Figure 22: Comparison of test accuracy across first ten and last five tasks on setups A and C

In Figure 22, we observe a clear trend where final accuracies on the last five tasks are consistently higher than those on the first ten tasks. This pattern is present

across both the C and A experimental setups. Thus, it cannot be attributed to the mid-training unlearning event introduced in C; rather, it appears to be an inherent feature of continual learning dynamics, which we discussed in Section 7.5.

Across algorithms, we observe that the final accuracies on the last five tasks remain broadly similar between A and C, with differences typically within a few percentage points and within the observed standard deviations. This suggests that the unlearning phase introduced in C does not substantially degrade the model's ability to subsequently learn new tasks.

However, we must not completely dismiss variations, as there are visible differences between A and C for some methods, such as RL-GEM and NegGEM, where last task accuracies in C are slightly lower compared to A. These differences still remain within expected variance bounds and from Figures 21 and 20, we see strong $1-L_{\text{ret}}$ and $1-L_{\text{res}}$ scores achieved by these methods indicate that their underlying retention and unlearning behaviours remain highly effective.

Overall, the results indicate that our continual unlearning does not significantly impair future learning capacity across the algorithms tested. These findings are encouraging, as they suggest that continual learning frameworks can incorporate unlearning phases without critically sacrificing the ability to absorb future tasks.

7.5.4 Forgetting Inbetween Learning (D)

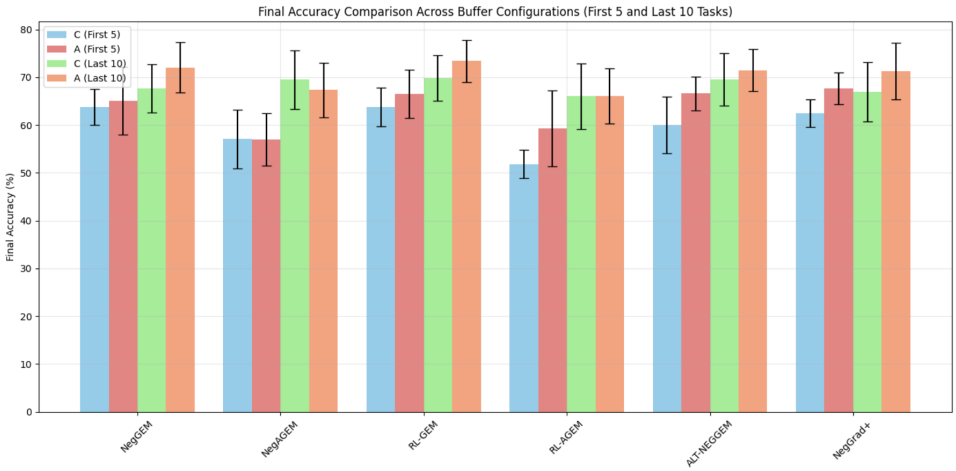


Figure 23: Comparison of test accuracy across first five and last ten tasks on setups A and D

Sequence D was inspired by Sequence C, but is designed to simulate the case

where a deletion request arrives earlier during the continual learning process. The aim was to evaluate whether earlier unlearning requests would introduce longer-term effects on the system's learning dynamics.

As shown in our results, the overall performance trends in D remain highly similar to those observed in C. However, we again note that for some algorithms, accuracies are slightly higher in the A scenario compared to D. While these differences generally fall within the expected standard deviations, the fact that the degradation predominantly favours the A runs suggests there may be a subtle longer-term impact of mid-stream unlearning on learning stability.

That said, performance on later tasks remains robust across all methods, and we observe no clear evidence that early unlearning adversely affects the model's ability to learn future tasks. This indicates that, under the current setup, unlearning does not lead to significant long-term degradation.

7.5.5 Random One-Shot Forgetting (E)

The random forgetting test best estimates a real world scenario, wherein one of the batches of data is found out to be poisoned long after learning that task. All algorithms achieve some form of forgetting on the randomly selected task.

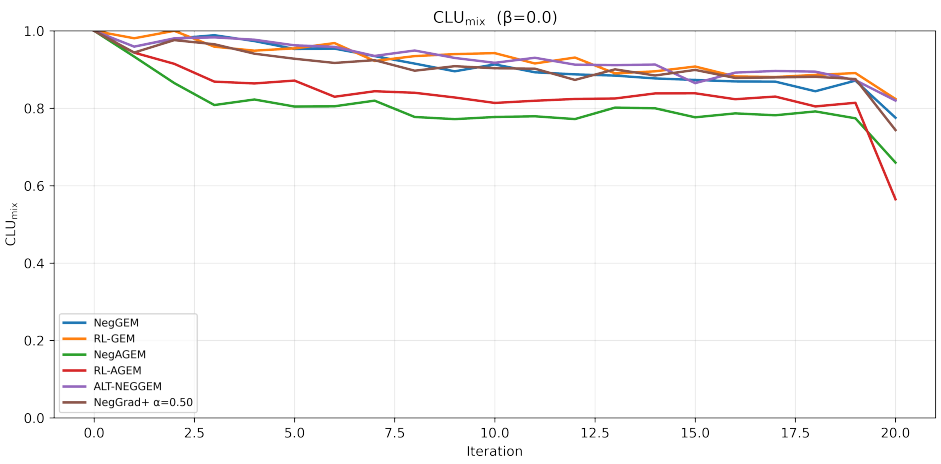


Figure 24: CLU_{mix} scores for task sequence E.

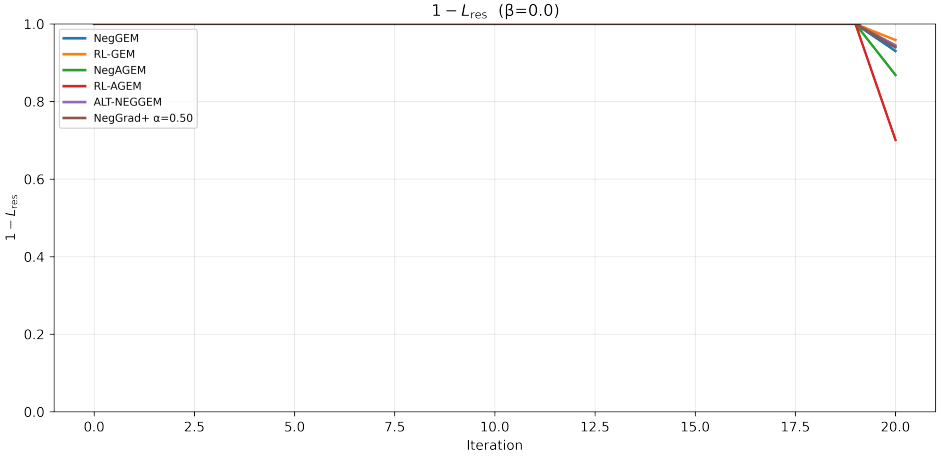


Figure 25: $1 - L_{\text{res}}$ scores for task sequence E.

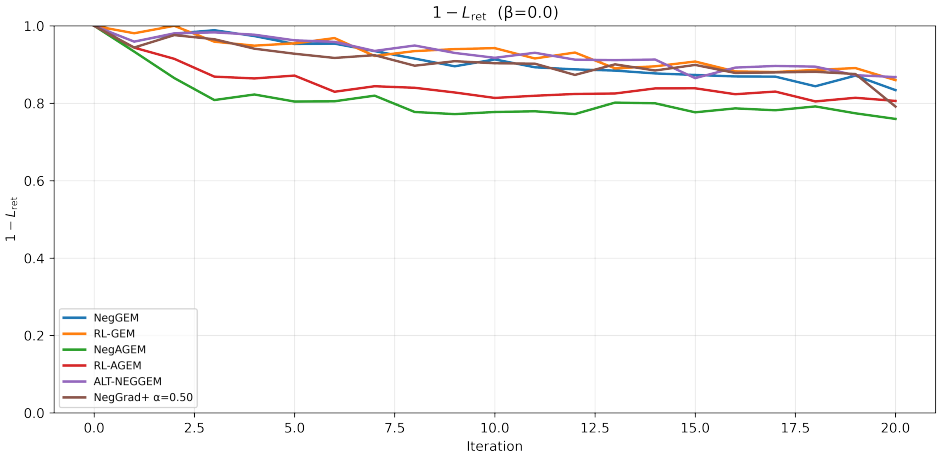


Figure 26: $1 - L_{\text{ret}}$ scores for task sequence E.

The continual learning phases occurs as normal until we proceed with the random forget sequence. For ease of reading we provide the test set task accuracy differences on the training iteration both before and after unlearning.

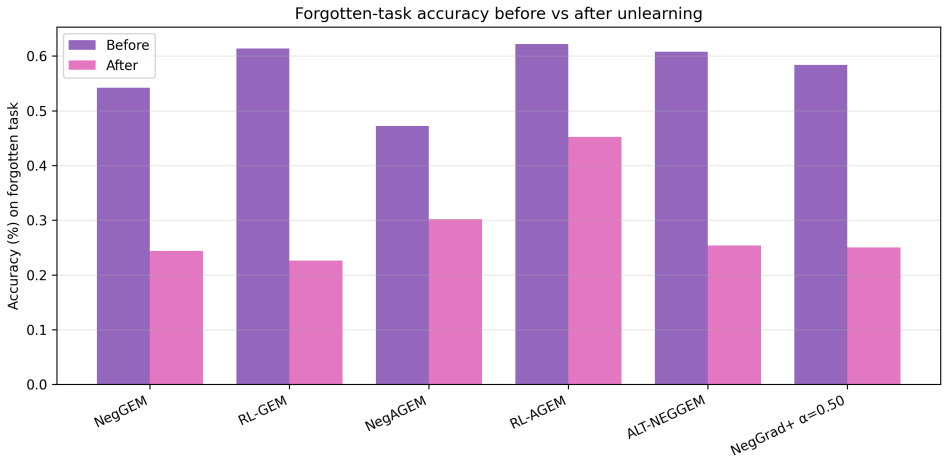


Figure 27: Task forget set accuracy before and after unlearning the random task per algorithm.

We observe that all algorithms except for the AGEM variants are able to achieve forgetting to the point where it could be considered random guessing $\tilde{2}0\%$. The weak gradient update of RL-AGEM leads to its poor forgetting ability on the random task test.

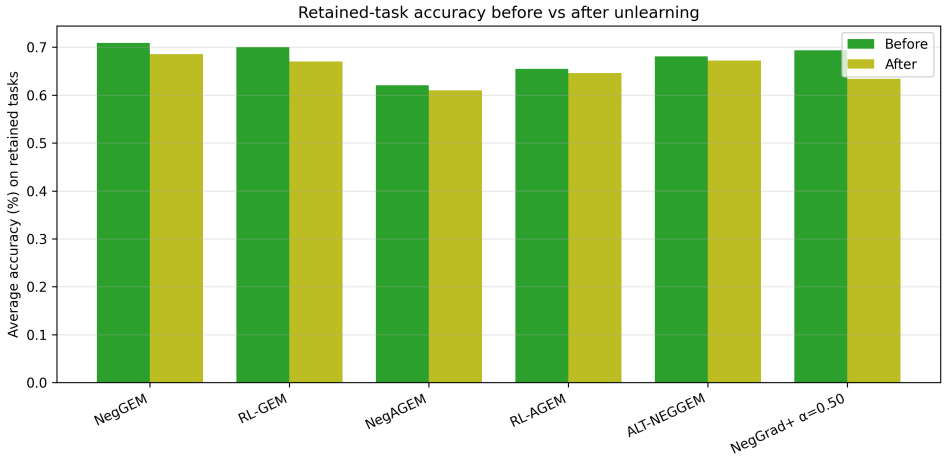


Figure 28: Task average retain set accuracy before and after unlearning the random task per algorithm.

Finally, in the vast majority of cases the drop in the average of retain set accuracy after the unlearn step was between 2-4% suggesting that in some cases, such as for NegGEM, when the number of observed tasks is large and we decide to unlearn one of them using NegGEM, then it becomes significantly harder to ensure that the negative gradient GEM constraint holds for all other task gradients.

7.5.6 Reinserting Forgotten Tasks (F)

Task sequence F is one of the two tests that we empirically analyse in order to demonstrate the presence of any residual affects of performance after a series of learning and unlearning requests. The task sequence F is as follows: Learn 10 tasks, Forget the Most recent 5 tasks, Learn 10 new tasks, and then relearn the forgotten 5 tasks. The primary objective of this sequencing is to answer the following question. If we unlearn a task, are there any residual effects when relearning those tasks at a later time?

In order to answer this question, we kept track of the initial accuracies of each task in the forgotten batch against the initial relearned accuracies and averaged this over 3 runs. To be more specific, consider task number x is first learned at time step x_l, then our initial accuracy of task x is the accuracy of that task at end the end of time step x_l. Building on this scenario, consider task number x has now been forgotten at time step x_f s.t. $x_f > x_l$ and has been relearned at time step x_r s.t. $x_r > x_f$ then task x's initial relearned accuracy is the accuracy of task x at the end of time step x_r. We further keep track of the forgetting accuracies that is defined as the average accuracy of our forget set directly after forgetting all of those tasks. Our results of this are found in Figure 29.

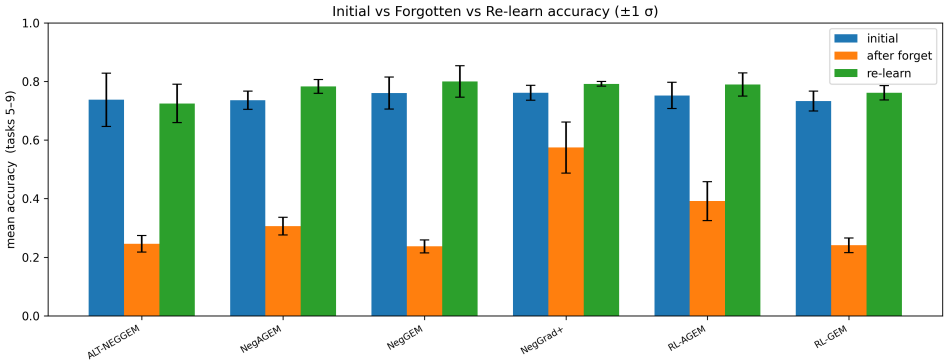


Figure 29: Comparisons of the average initial accuracies, average forgetting accuracies and average initial relearned accuracies of the five tasks that are learned, forgotten then relearned over 3 runs.

Figure 29 shows that all algorithms have at least as good initial relearned accuracy compared to their initial accuracy. This is particularly impressive across ALT-NEGGEM, NegAGEM, NegGEM, and RL-GEM which all have a desirable low forgetting accuracy coupled with a high initial relearned accuracy demonstrat-

ing that the model is capable of relearning a completely forgotten task to at least initial accuracy. Additionally, there is a slight trend that shows that after unlearning tasks, the model is put in a state whereby it can easily pick up the task and has incentives to have a very high accuracy possibly suggesting a slight forward transfer w.r.t. unlearning; however, this trend is not convincing enough for us to concretely make this claim.

We further study the overall performance and retention performance of our models during the life of task sequence F. After reintroduction of forgotten tasks at the end of the training, Figure 31 shows that GEM-based algorithms (NegGEM, RL-GEM, ALT-NEGGEM, NegGrad+) have retention levels close to 1, indicating effective initial learning ability and reversibility. In contrast, AGEM-based algorithms, particularly RL-AGEM, struggle to regain their original retention levels. We can also see that NegGrad+ overall performance is relatively poor from the CLU metric in Figure 30 and struggles in this particular test due to the performance being very susceptible to the tuning of alpha amongst different scenarios as mentioned in Section 7.5.1.

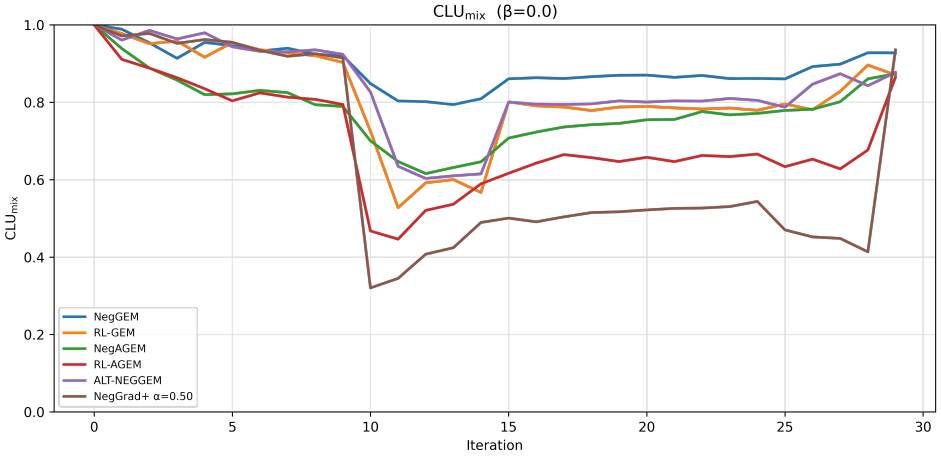


Figure 30: CLU_{mix} scores for task sequence F: Performance when previously forgotten tasks are reinserted.

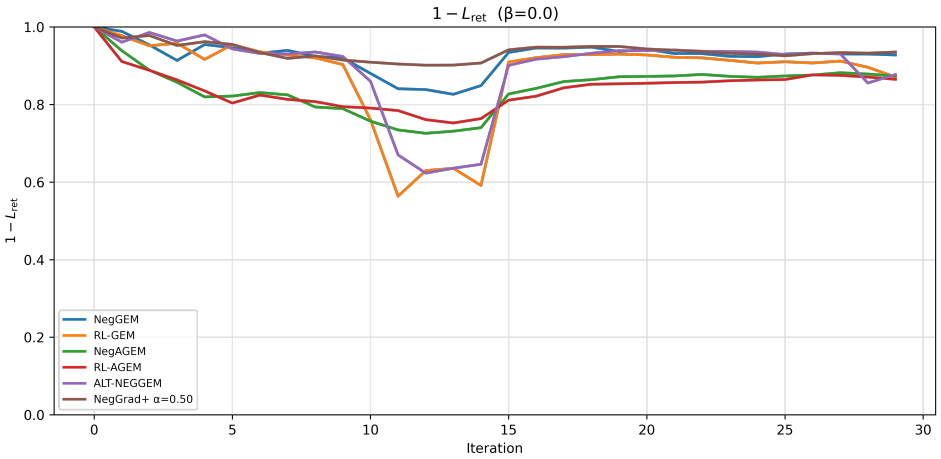


Figure 31: $1 - L_{\text{ret}}$ scores for task sequence F: Knowledge retention during the full forget-reinsert cycle.

7.5.7 Reinserting Forgotten Tasks (G)

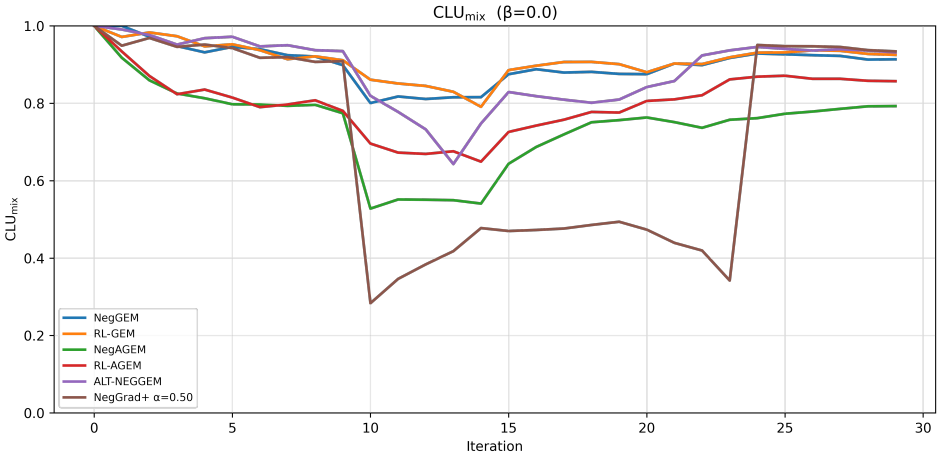


Figure 32: CLU_{mix} scores for task sequence G: Performance when forgotten tasks are immediately reintroduced.

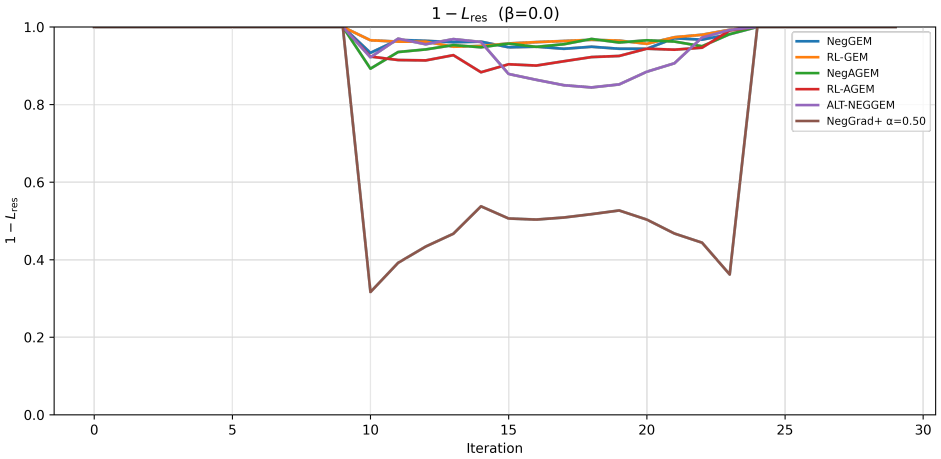


Figure 33: $1 - L_{\text{res}}$ scores for task sequence G: Forgetting effectiveness with rapid task reintroduction.

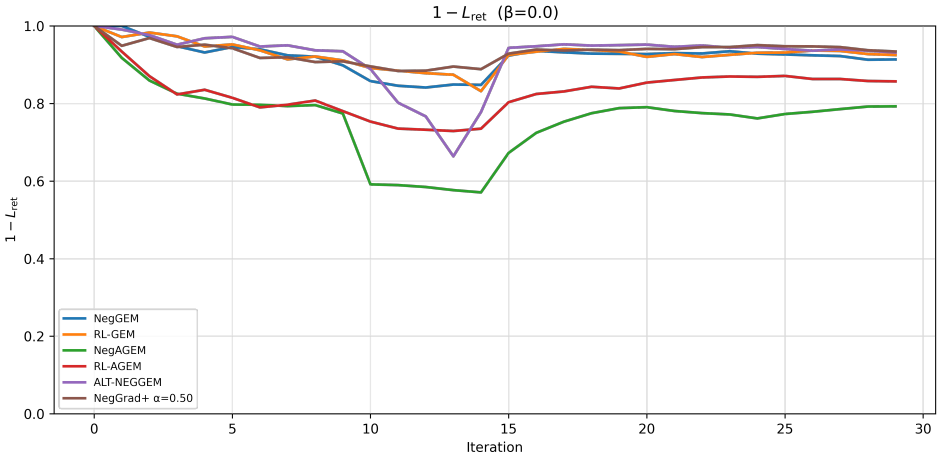


Figure 34: $1 - L_{\text{ret}}$ scores for task sequence G: Knowledge retention during rapid task reinsertion.

In Fig. 33 the GEM and A-GEM variants drive the forget-set metric to its ceiling ($1 - L_{\text{res}} \approx 1$), showing that their projection step all but eliminates residual signals from the deleted tasks. NegGrad$^+$ does not keep pace: a single deletion already lowers $1 - L_{\text{res}}$ sharply, confirming that an unchecked sign-reversal is insufficient once the stream advances.

The retention plot in Fig. 34 completes the picture. Constraint-aware updates (NegGEM, RL-GEM, NegGrad$^+$) hold $1 - L_{\text{ret}} \geq 0.8$ throughout, whereas the average-gradient variants (RL-AGEM, NegAGEM) sacrifice roughly ten additional percentage points—exactly the loss expected when the projection cone is replaced

by a single averaged constraint.

Because tasks 5–9 are immediately re-introduced after deletion, the sequence probes erasure and forward transfer at once. The results make the point succinctly: *structured unlearning matters*. Purely negated gradients erase less and relearn poorly; projection-based updates forget decisively while leaving the model ready to relearn the same content when required.

Taken together with the outcomes on all other sequences (A–H), the G and A-GEM curves underline a broader rule: forgetting, retention, and forward transfer form a three-way trade-off that must be managed jointly. Any method that neglects one leg of this triangle—whether by loosening constraints or by ignoring future plasticity—will falter once the task stream demands all three properties at once.

7.5.8 Average Accuracy of Last 5 Tasks (G-A (Learning Portion))

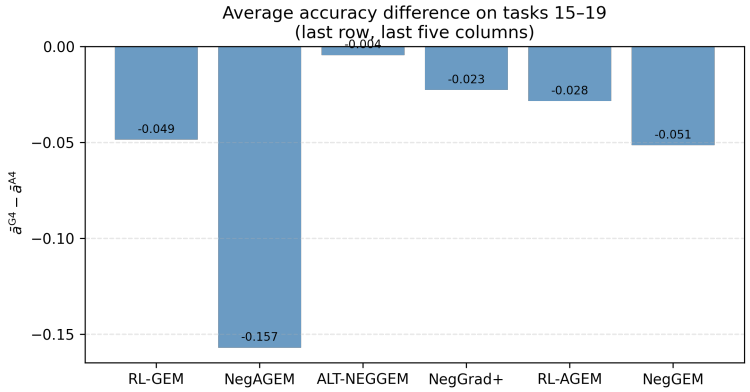


Figure 35: The difference in average accuracy of the last 5 tasks of G and the continual learning portion of A

To extend on this notion of future learning being of importance we devised and experiment to support the following: One of the key aspects of unlearning that we wanted to measure was the difference between a model that had gone through many spaced *learn–unlearn* cycles compared to a case where only continual learning occurs. From Figure 35 we observed the average test accuracy difference between the last 5 tasks learned on the learning portion of A and the entire portion of G sequences. We observed that certain algorithm variants such as ALT-NEGGEM are superior in the case of forward transfer for learning the final tasks compared to others. This is mainly evident in the fact that for RL-GEM and NegGEM their

differences are roughly identical at -0.050. This is effectively evidence that shows that the rate of forward transfer in later learning can be massively hindered by the constraints specified in the continual learning and unlearning algorithm. Despite this shortfall, a decrease of -0.050 is relatively small and suggests that a framework as such is still a viable option. Interestingly, the reduced memory buffer size has a large detrimental impact in this test case, possibly raising questions to the viability of extremely small buffers or averaged gradient projection methods.

7.5.9 Spaced Periodic Forgetting (H)

Task sequence H is the only testing scenario where we aim to discover if there are any affects on the models performance based on certain requests engineered at specific times. More specifically, Task sequence H aims to discover if there is a relationship between when a task is learned and how well we retain the tasks after forgetting it. To simulate this test, we learn all tasks, then forget tasks numbers of a multiple 5 starting from 20 going backward. An idealistic result would be performant forgetting coupled with high retention scores indicating that the time in which we unlearn tasks does not negatively affect the model.

Figures 36, 37, 38 show our CLU, $1 - L_{res}$ and $1 - Lret$ metrics over time respectively. Zooming in on our $1-Lret$ metric we can see that there is no downtrend in our retain performance, and instead there is a minor dip after the first unlearn step at iteration 21 in only ALT-NegGEM, RL-AGEM and NegAGEM. This would mean nothing unless the forgetting performance is also good. Honing in on our $1-Lres$ metric we also see exceptional performance with the lowest of the algorithms still surpassing 0.8. Thus, we can conclude that unlearning certain tasks with respect to time has no noticeable affect compared to unlearning an arbitrary task.

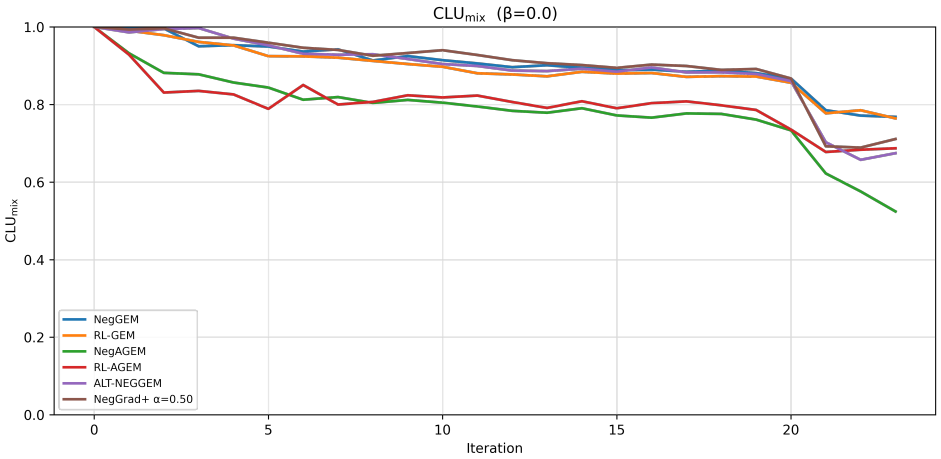


Figure 36: the $\mathrm{CLU}_{\mathrm{mix}}$ score of all algorithms over the life span of task sequence H

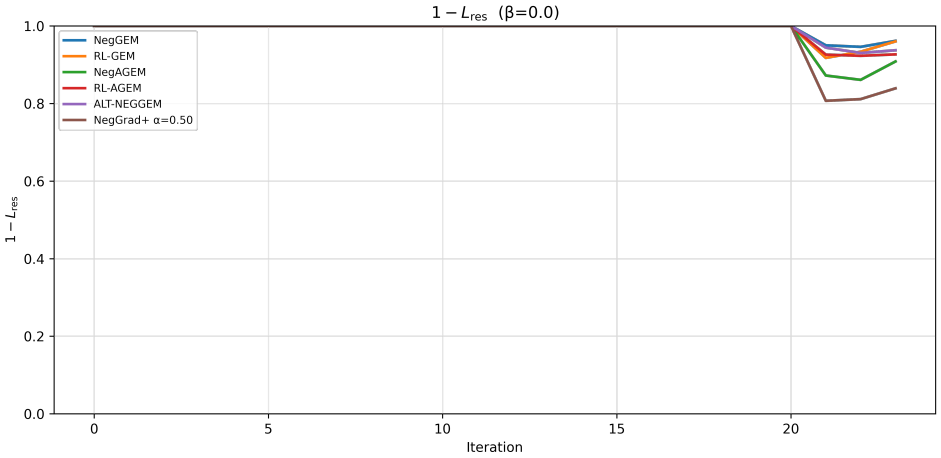


Figure 37: $1 - L_{\mathrm{res}}$ scores for task sequence H

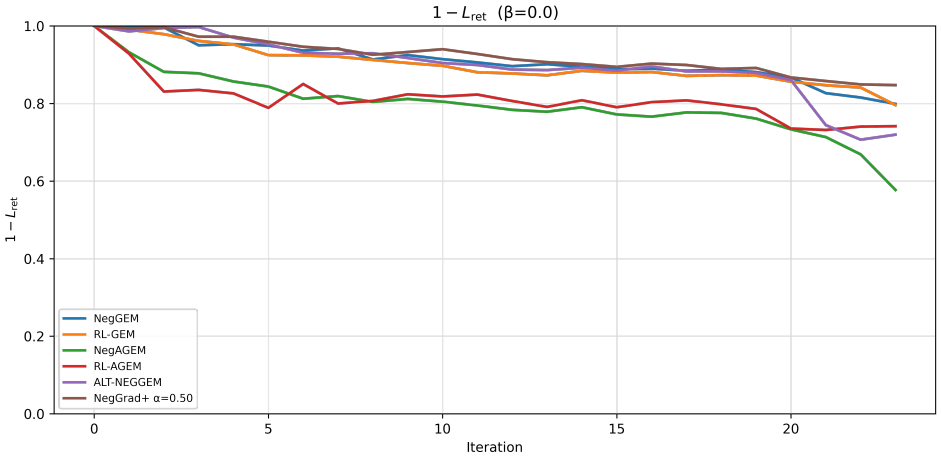


Figure 38: $1 - L_{\mathrm{ret}}$ scores for task sequence H

7.6 MIA Testing Results

In this section, we extend the application of MIA beyond its traditional use in unlearning evaluation and investigate its behaviour during continual learning. Our hypothesis is that the continual learning process—particularly task-wise learning and memory-based replay—has non-trivial effects on the model's vulnerability to MIAs.

Specifically, we analyse the success rate of Membership Inference Attacks (MIAs) over iterations as a model is trained sequentially across ten tasks. For each task, we evaluate MIA success on two datasets:

- The full training set of that task (all samples originally seen during training),
- The reduced buffer subset (the small set of replayed examples stored during continual learning).

The model is trained over ten iterations (one per task), with a fixed replay buffer updated after each task using the ideal hyper-parameters per algorithm identified earlier. To capture how continual learning impacts privacy risk at different stages, we track the MIA success rate on tasks 1 through 8 individually during every learning step.

Importantly, for a given task t, meaningful MIA results can only be obtained starting from iteration t onwards:

- Before the task is learned, its samples have not been exposed to the model, resulting in random guessing by the MIA (AUC ≈ 0.5).
- For the buffer, prior to learning, the corresponding buffer entries are uninitialised, artificially leading to trivial MIA attacks (AUC ≈ 1).

As such, for each task, MIA evaluations begin at the iteration when the task is first introduced into training, and are tracked up to the end of the learning phase (iteration 10). Metrics reported are *Attack Accuracy* and AUC. We primarily focus on AUC (Area Under the Receiver Operating Characteristic Curve) as our evaluation metric. It captures the overall separability between member and non-member examples across all possible decision thresholds, offering a threshold-independent evaluation of attack strength. As such, AUC better reflects the underlying privacy leakage characteristics of the model.

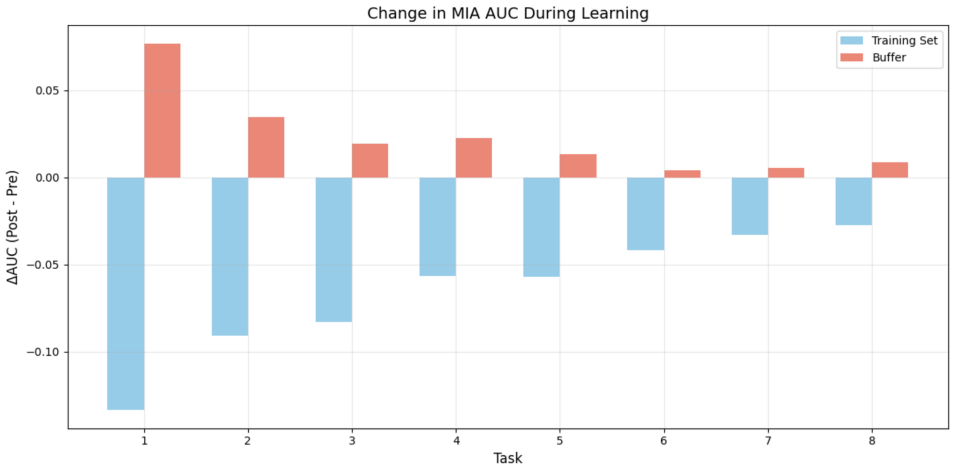


Figure 39: Change in MIA AUC across iterations for Tasks 1–8.

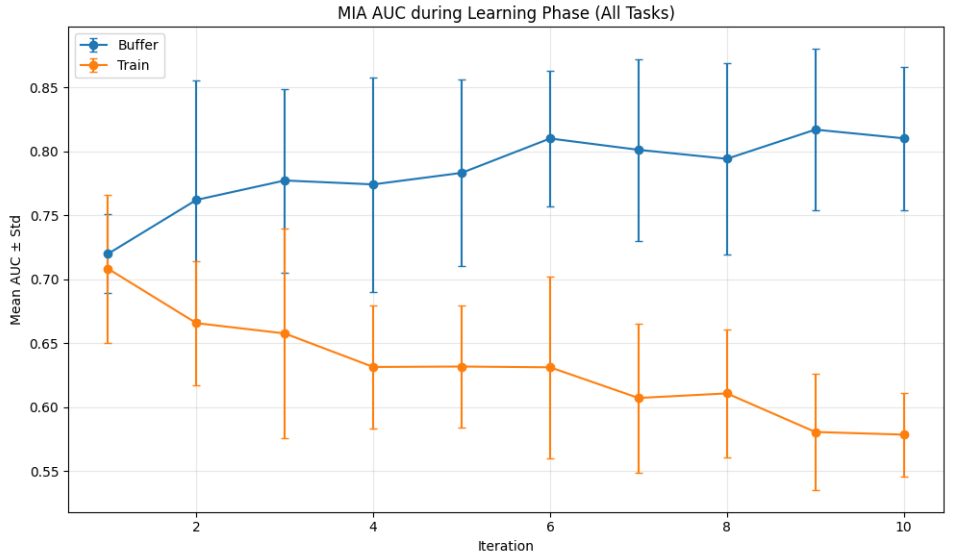


Figure 40: Mean and standard deviation of MIA AUC over iterations across tasks.

From these results we see a key trend of MIA success on the training set consistently declining over time. Following the initial task training, models exhibit high vulnerability (attack accuracies around 70%). However, as continual learning progresses with exposure to new tasks, MIA success rates on the training set approach random guessing (approximately 55%), and AUC scores fall substantially.

Conversely, MIA success on the memory buffer samples increases over iterations. Due to repeated replay, samples in the buffer become more distinguishable to the attack model, leading to a noticeable rise in MIA attack accuracy and AUC over time.

Importantly, this implies that CL allows models to emulate high overall dataset performance while exposing only a small subset (the memory buffer) to higher privacy risk. Thus, for effective unlearning, we must focus not only on global training set metrics but explicitly target and mitigate MIA exposure within the memory buffer, as low MIA vulnerability on the entire training set can create a false sense of security.

This motivates our next analysis, which is a comparison of unlearning performance across algorithms, evaluated both on full training sets and learning buffers.

In this experiment, we use the continually learned model and unlearn tasks sequentially from Task 10 down to Task 1. At each unlearning step, we evaluate the Membership Inference Attack success on previously learned tasks. Specifically, for each task, we record the MIA results at the iteration immediately following its unlearning. The reported AUC values are obtained by averaging across Tasks 1–8, allowing us to summarise overall algorithm performance on both the full training set and the memorised replay buffer.

Algorithm	**Train AUC**	**Buffer AUC**	**Retain Acc.**	**Forget Acc.**
NegGEM	0.499 (±0.022)	0.533 (±0.040)	61.5	24.2
NegAGEM	0.506 (±0.040)	0.766 (±0.053)	56.8	25.3
RL-GEM	0.489 (±0.025)	0.522 (±0.052)	59.2	22.5
RL-AGEM	0.523 (±0.016)	0.825 (±0.037)	56.8	23.0
ALT-NegGEM	0.488 (±0.026)	0.532 (±0.041)	62.6	24.9
NegGrad+	0.511 (±0.026)	0.570 (±0.036)	57.5	22.5

Table 3: Comparison of MIA final AUCs (mean ± std) and overall algorithm performance (retain/forget mean accuracies).

Following the averaged MIA results, we further contextualise the performance of different algorithms by analysing their respective retain and forget set accuracies after continual learning and unlearning (Table 3).

Across all algorithms, the mean retain accuracies are relatively similar, except for the AGEM-based variants (*RL-AGEM* and *RL-GEM*), which consistently achieve lower retain performance, consistent with our earlier findings. Furthermore, the mean forget set accuracies are low and comparable across all methods (typically around 22%–25%). This consistency is important: it ensures that differences observed in MIA success are not simply due to variations in task retention or forgetting. In particular, if one algorithm retained substantially more information than

others, its higher MIA vulnerability would be expected and thus less meaningful to compare.

In terms of final MIA results, all algorithms achieve strong privacy performance on the full training set after unlearning. This is in line with the earlier findings that Continual Learning combined with replay buffers is effective at mitigating membership leakage from the overall dataset. Notably, models are able to maintain generalisation performance.

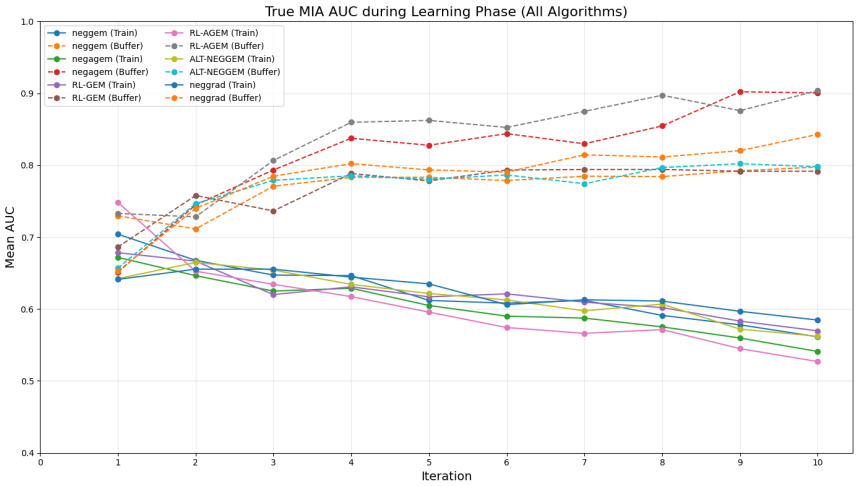


Figure 41: MIA AUC over training iterations for each algorithm.

However, a significant divergence emerges when examining the memory buffer. The AGEM-based methods (*RL-AGEM* and *Neg-AGEM*) perform substantially worse than others, exhibiting much higher final MIA AUCs on the buffer samples. This likely stems from their significantly smaller buffer sizes. With fewer examples replayed, the model overfits more heavily to the limited memorised samples, making them more distinguishable to MIA attacks and harder to effectively unlearn. This trend is clearly illustrated in Figure 41, which shows that AGEM variants have the highest buffer MIA AUCs post-training.

Among the remaining methods, our proposed approaches (*NegGEM* and *ALT-NegGEM*) achieve notably better unlearning performance compared to the *Neg-Grad+* baseline, particularly on buffer samples (AUC ≈ 0.53 vs. 0.57). This validates our hypothesis that leveraging information gathered during learning enables more targeted and effective unlearning than applying negative gradient updates indiscriminately.

8 Results Evaluation

In the previous chapter, we evaluated continual unlearning algorithms under a wide range of scenarios designed to mimic real-world deletion requests: long sequences of contiguous forgetting, volatile learn–forget cycles, mid-stream deletion events, reinsertion of previously forgotten tasks, and targeted temporal forgetting. These tests allowed us to draw several key lessons about the behaviour of continual unlearning systems and the interplay between learning and forgetting dynamics.

Stress Testing Insights: Stress testing across both the long-horizon sequence A and the rapid *learn–forget* sequence B reveals several broad insights. First, despite the challenge of forgetting large amounts of knowledge, all methods successfully pushed forgotten-set accuracies towards chance while preserving significant information about retained tasks, achieving composite $\mathrm{CLU}_{\mathrm{mix}}$ and CL-ToW scores above 0.6 for most of the unlearning curriculum. This highlights that constraint-based updates (e.g., GEM-style projections) remain effective even under extreme deletion pressure.

Second, curriculum volatility plays a major role: when tasks are erased immediately after being learned (as in sequence B), the models experience sharper drops and rebounds in retention compared to a stable, contiguous learning phase. This instability shows that high-volatility regimes amplify the tension between retention and forgetting, suggesting a need for either larger, more diverse replay buffers or unlearning techniques that explicitly encourage confusion to smooth model updates.

Third, compute budgets place practical constraints on algorithm choice. Methods that rely on small gradient steps, extra projection solves, or aggressive sparsification achieve superior forgetting–retention trade-offs, but incur much higher computational cost. In realistic deployment, efficient variants that offer slightly lower peak scores but faster turnaround may be preferable to full retraining or heavily constrained GEM-like approaches.

Mid-Stream Forgetting and Longer-Term Effects: Testing sequences C and D were designed to assess whether unlearning during training introduces lasting damage to the model's ability to learn future tasks. Our results show that, while mid-training deletion events slightly reduce accuracies compared to a purely con-

tinual learning setup, the differences are generally within the margin of standard deviations. Notably, models still retain high forward plasticity, suggesting that careful unlearning does not block future learning altogether.

However, subtle trends emerged: across multiple algorithms, the setups with mid-training unlearning consistently showed marginally lower accuracies compared to purely continual learners. Although the gaps were small, the fact that the degradation was systematically in the same direction suggests that mid-stream unlearning may introduce slight long-term stability impacts, particularly for methods that tightly couple gradient constraints to memory updates. These findings reinforce that while mid-training deletion is survivable, it is not entirely without cost—and algorithms should be designed to minimise the disruption to future learning when unlearning occurs.

Relearning Forgotten Tasks: Sequences F and G tested whether tasks that were previously unlearned could be reintroduced and relearned effectively. We found that all algorithms were capable of relearning previously forgotten tasks to at least their original initial accuracies, often exceeding them slightly. GEM-based variants such as NegGEM and RL-GEM showed particularly strong reversibility, maintaining high retention and demonstrating that constraint-based approaches allow models to "forget cleanly" without damaging their ability to recover forgotten information later.

Interestingly, in scenarios where tasks were forgotten and immediately relearned (sequence G), caused minor negative affects on the performance of learning new tasks showing a particularly large weakness in AGEM based approaches raising a question to the viability of averaged gradient based approaches. Despite this, the minor negative affects are not significant enough to conclude that GEM-based approaches perform worse on new tasks after series of learning and unlearning.

Membership Inference and Privacy: A critical part of our evaluation was examining privacy leakage through membership inference attacks (MIA) both during continual learning and after unlearning. We observed that continual learning itself progressively reduces MIA vulnerability across the full training set, as the model's exposure to diverse tasks dilutes per-sample memorisation. However, mem-

orised buffer samples—replayed during continual learning—remain more vulnerable to MIAs over time, highlighting the need to pay special attention to buffer composition during both learning and unlearning.

Among the unlearning methods tested, structured approaches like NegGEM and ALT-NEGGEM achieved lower MIA success rates compared to the NegGrad+ baseline, especially when focusing on buffer samples. Projection-based and learning-aware methods were better at erasing sensitive information without harming generalisation, reinforcing that informed, structured unlearning is necessary to maintain privacy guarantees in continual settings.

Final Reflection Across all experiments, one conclusion becomes unavoidable: learning and unlearning must be designed together, not separately.

Successful unlearning is not just about cancelling out past gradients or deleting memory buffers—it depends crucially on the structures laid down during initial learning. Algorithms that enforced structured learning updates (such as GEM-style constraints) created conditions where unlearning could be performed selectively, precisely, and efficiently, preserving the ability to learn new tasks and protecting privacy.

By contrast, methods that treated learning and unlearning as independent phases—first memorising freely, then applying naive erasure techniques—struggled with residual memorisation, poor future learning capacity, and unstable forgetting.

Thus, our results show that continual unlearning systems must adopt a holistic design philosophy: the way a model is trained must explicitly prepare it for the possibility of future unlearning. Learning and unlearning are two sides of the same coin, and only by crafting them to work synergistically can we achieve models that learn continuously, forget selectively, and do so with efficiency, robustness, and privacy.

9 Project Management

This section details the resources and tools employed, the project management methodologies adopted, the strategies for risk management and mitigation, and the role and notable impact of meetings on project velocity.

9.1 Resources

9.1.1 Datasets

We selected benchmark image classification datasets that are widely used in machine learning research to ensure comparability with prior work. CIFAR-10 is commonly used to evaluate general-purpose classification algorithms and is particularly useful for analysing unlearning behaviour due to its relatively low complexity and well-structured class boundaries. Introduced by Krizhevsky et al., CIFAR-10 consists of 60,000 colour images of resolution 32×32 pixels, divided into 10 mutually exclusive classes (e.g., airplane, automobile, bird), with 50,000 training images and 10,000 test images (29). All classes are balanced, each containing exactly 6,000 images.

CIFAR-100, introduced in the same work, presents a more fine-grained classification challenge. It comprises 100 classes grouped into 20 superclasses, with 600 images per class—500 for training and 100 for testing—while maintaining the same image format and total dataset size. Due to its larger label space and reduced per-class sample count, CIFAR-100 is significantly more complex.

For initial experimentation, debugging, and proof-of-concept testing, we used CIFAR-10 due to its lower computational demands. Once the training and unlearning pipelines were validated, all final experiments and evaluations were conducted on CIFAR-100 to assess performance in more challenging situations.

9.1.2 ResNet-18

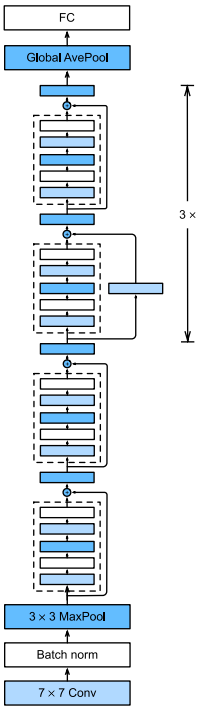


Figure 42: Resnet-18 Architecture Diagram: (30)

Our model architecture is based on ResNet-18, a widely adopted convolutional neural network introduced by He et al. as part of the Residual Network (ResNet) family (31). ResNet-18 consists of 17 convolutional layers and one fully connected output layer, structured into four stages of residual blocks. Each block includes skip connections that perform identity mapping, enabling gradients to flow more easily through the network and addressing the vanishing gradient problem common in deeper architectures.

Residual networks have become a standard backbone in image classification research due to their simplicity, efficiency, and strong performance on benchmark datasets. In particular, ResNet-18 is frequently used in continual learning and machine unlearning experiments involving the CIFAR datasets. Its relatively shallow depth, compared to ResNet-50 or ResNet-101, provides a well-balanced trade-off between model capacity and computational cost. This project was going to require the use of DCS hardware, and our own personal devices. Using a large model would have meant longer testing times, making iteration slower. Since the project was

never limited by the use of ResNet-18, that was the only model used.

In our implementation, we adapted the final classification layer to output logits corresponding to the number of classes in each task. This allowed us to apply task-specific masking during inference, restricting predictions to relevant class indices in continual learning settings. The remainder of the ResNet-18 architecture was used without modification.

9.2 Development Tools

Python remains the dominant language in machine learning research due to its extensive ecosystem, and integration with high-performance computing tools. Hence, the experimental aspects of this project were implemented in Python 3.

One of the key focuses of this investigation was the performance of the algorithm in terms of training and retraining time. As we moved toward optimisation on larger datasets such as CIFAR-100 (27), program runtime became an important factor limiting the ability to explore different branches of the project. As a result, accelerating our algorithms became necessary in the later stages of development to keep compute times on the Department of Computer Science Batch Compute system feasible (32).

9.2.1 Libraries

PyTorch

PyTorch provides a comprehensive toolkit for deep learning, including modules for model construction (torch.nn), loss functions, optimisers, and GPU acceleration via torch.cuda (33). In this project, it was necessary for the development of a fully custom neural network training loop tailored to continual learning and unlearning.

Our model class extended torch.nn.Module, wrapping a ResNet-18 backbone with logic to handle task-specific class masking. Backward passes relied on manual loss computation and the provided .backward() to maintain control over gradient flow.

Per-task gradients were stored in preallocated tensors using PyTorch's.grad attributes and manipulated directly to enforce constraints used by our algorithms. Updates were applied through optimisers such as Stochastic Gradient Descent (torch.optim.SGD)

(33) after gradient projections ensured compliance with memory-based constraints.

PyTorch's tensor operations were central to all data handling, from reshaping and batching to indexing and memory buffer updates. Its GPU integration allowed for scalable training across multiple runs and large datasets.

In general, PyTorch was critical in allowing low-level control, efficiency, and modularity required to build and evaluate our continual learning and unlearning framework.

QuadProg

QuadProg is a Python wrapper for a Fortran-based solver that handles quadratic programming problems with linear constraints. In this project, it was used to compute gradient projections for the GEM algorithm by solving the optimisation problems described. QuadProg's speed and numerical stability of QuadProg made it suitable for applying these projections at each training step without introducing significant overhead.

Matplotlib

Matplotlib was used for visualising training and evaluation results across tasks and runs. It enabled us to create line plots tracking metrics such as accuracy, confidence, and membership inference attack performance over time. These plots were key to analysing trends in the behaviour of algorithms across different tasks and buffer configurations.

9.2.2 Hardware

Partition Specification	**Gecko**	**Falcon**
CPU	Dual AMD EPYC 7443	Intel Core i5-11500
Core Count	48 Core/96 Thread	6 Core/12 Thread
Memory	512GB (Limit 128GB)	64GB
GPU	3 x Nvidia A10	1 x Nvidia A5000
Video Memory	24GB	24GB

Table 4: DCS Gecko vs. DCS Falcon

We opted to use GPU-accelerated compute nodes ('Gecko') equipped with Nvidia A10 accelerator cards. These provide Tensor Core support for general matrix multiply (GeMM) operations, a large component of deep learning workloads. Prior research reports up to 5× speedups using Tensor Cores versus standard CUDA cores (34). This optimisation allowed us to conduct large-scale tests within practical time constraints and enabled more extensive evaluation of algorithm variants. This became especially important in the final weeks of testing, when we were generating a bulk of our final results.

A DCS large data store was also provisioned for the group after discussion in one of the first setup meetings. At the early phase of the project, potential datasets were discussed, including subsets of ImageNet. In that case more storage would have been needed than the default DCS user areas. Ultimately, these large datasets were never used, so the large data store was never used.

9.3 Version Control and Collaboration

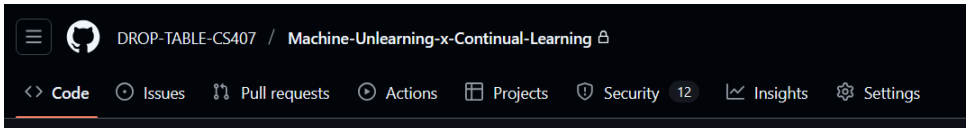


Figure 43: The GitHub organisation set up by the group, with the resources that that gave us access to.

Each person in the group had their own workflow for working on the code, mostly using Visual Studio Code. A way of bringing the code together was important. Git was used for version management and for enabling parallel work. GitHub was used to store a remote copy of the code repository, letting us work on it on our own devices. Use of GitHub also gave access to other tools provided by the service. To this end, a GitHub Organisation was set up containing every member of the group. Each member would be encourage to work off of a separate branch for their own testing and implementation steps. In cases where a task was being worked on by multiple people, the same branch would be used on separate devices. When a task was finished, and could be merged to the main branch, a pull request would be opened on the main branch of the code repository. For code quality assurance, a merge could not be approved by the person making the request. This meant that all

code would have had at least two pairs of eyes on it before being added to the main code base. For obvious reasons, this helped prevent bugs and encouraged good code quality. Git also provides a log of code edits, additions, and deletions. By using the tools provided by Git, it was trivial to do code rollbacks on the rare occasions errors were observed.

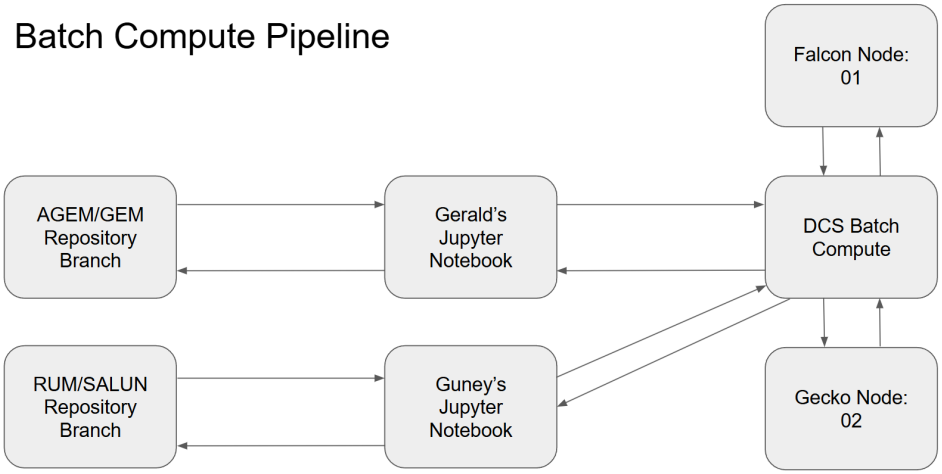


Figure 44: The Jupyter setup combined with GitHub version control from the term 1 week 10 presentation. Each member edits their own version of the Jupyter notebook during the exploratory stage and can pull/push from a central GitHub repository.

For remote meetings, and general communications, we used the online messaging software Discord. Discord provides text and audio channels, and supports file uploading. It also provides screen mirroring and file uploading, making it an effective way of working when not in the same room. We maintained separate text chats for different topics (Open pull requests, results, work required, etc.), and used the audio channels for meetings and discussions.

Notion (a general notes taking app) was used as a collective knowledge store and Kanban board. A simple system of "Not started", "In progress", and "Done" was used. When a certain task needed doing, a task was added to the board in the first column. Once somebody took the task, it would be assigned to them and placed in the second column. When finished, it would be moved to the third column. This gave a constant note of what needed doing and who was doing what. Outside of this, smaller tasks were managed informally through verbal agreements, with individuals taking responsibility for a task and presenting their progress at the following meeting.

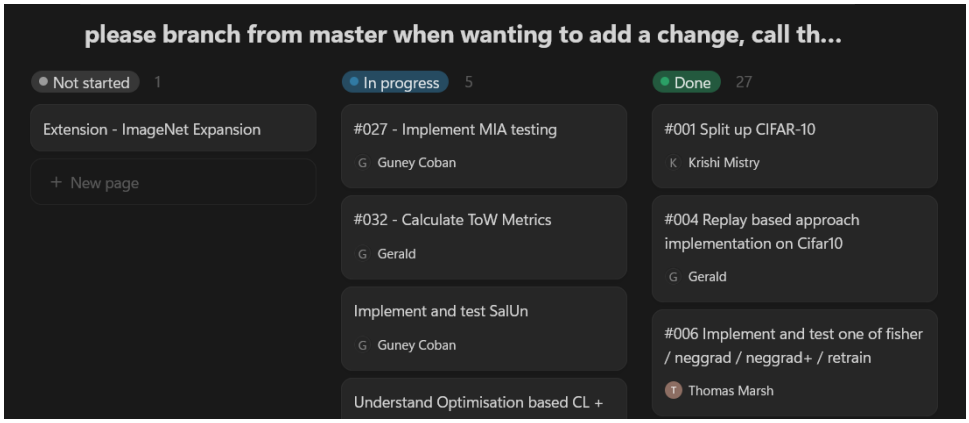


Figure 45: The Kanban board used for keeping track of larger pieces of work. By assigning names to work, we could keep track of who was doing what.

9.4 Project Management Methodology

This project is not a traditional software development project, instead being a hybrid between research project and software development. Any software developed was for the ultimate end goal of results. Regardless, good software development practices were important to ensure the codebase was high quality, and project management techniques were used to ensure the project as a whole went smoothly. We used Agile methods, specifically "Scrumban", a mix of scrum and Kanban.

Scrumban worked well for us; as a small and flexible team there was no need for detailed external knowledge stores or documents laying out requirements. If we needed to discuss someone else's work, it was trivial to sort out a time for them to explain it to us. Decisions could be made as a group through a simple vote without a complex organisation system of project managers and dedicated teams.

A more traditional software method such as waterfall would have been wholly unsuitable because of the research aspect. We did not know what the results would be going into this project, and we needed the flexibility to change focus. As mentioned in our original specification "It would be unproductive to prevent ourselves from pursuing something promising because it wasn't in the plan". However, it was important to ensure that loose requirements did not become low requirements. To this end, effort was spent on ensuring we matched an expected project velocity. This was achieved through regular meetings with our supervisors in the early parts of the project, and regular meetings with each other in later parts of the project. This was not always successful in ensuring consistent velocity.

9.4.1 Development-to-Deployment Testing

Initial experimentation and debugging were conducted using Jupyter notebooks due to their interactive nature, allowing section-by-section execution and variable retention across cells. This facilitated iterative development and model inspection in a single file. This approach worked well for small-scale testing but was problematic for larger datasets. Jupyter notebooks require a dedicated server, which was impractical to keep running on the batch compute system. Consequently, the testing workflow was migrated to stand-alone Python scripts, which in turn allowed multiple test files to be packaged into single SLURM jobs to be run on the batch nodes. This change in methodology allowed us to fully automate testing across different parameters and algorithms for long-running experiments and reduced the manual overhead of keeping Jupyter servers running.

9.4.2 Team Roles

While trying to ensure that everyone had a good understanding of the entire project was important for merging CL and MU concepts, it was inevitable that people would have to be put in charge of certain parts of the project. This was necessary for coding reasons - one person can iterate code they wrote faster than code someone else wrote, since they already know what the code does. Instead of clearly defined roles, these would become general focuses owned by these people.

Name	**Original Focus**	**Other Areas**
Guney Coban	Machine Unlearning	Memory Buffer Composition Research, MIA Metric Implementation
Thomas Marsh	Machine Unlearning	MU implementations, Project Management
Krishi Mistry	Continual Learning	CL Research, CL implementation, Testing Framework. Pytorch Expert.
Gerald Ramos	Continual Learning	Python CUDA and PyTorch Expert. Hyper-parameter Tuning. Testing Scripts.
Paul Joakim	Machine Unlearning	Metric Implementation, NegGrad, Specification Auditor

Table 5: General focus and other sections managed by each person on the team

These roles were more important to code ownership and general understanding of the topic, instead of being strict management roles within the group. This

had benefits and drawbacks. The lack of dedicated project management roles was because this project was a research project over a software development project. There was no cohesive structural requirement to the code - results mattered more - so it was reasonable to split code into chunks based on the testing being run. There was no need for a dedicated "UI", "Backend", or "Frontent" team as expected in other projects.

9.5 Risk Management and Mitigation

Within the specification, we identified several risks that could compromise the success of the project.

9.5.1 Loss of Access to Resources (Critical)

As mentioned, we required the DCS batch compute system to run larger tests. We expected that should this become an issue, it would massively impact the project as a whole. While the batch compute system helped us scale up experiments, we still had our challenges regarding compute time. Towards the end of the project, we were regularly running 12–24 hour jobs, meaning any failure would cost us an entire day of progress. An unexpected challenge came from the traffic on the nodes: many students were working on third-year projects, and the system was frequently at capacity. As a result, we often had to wait hours just to get our jobs scheduled. In many cases, we were forced to run smaller tests on slower fallback nodes like 'Falcon' and 'Eagle', which delayed results and limited how much testing we could do. This bottleneck made it harder to iterate quickly.

We expected this problem to be critical to the projects success, and through these issues, this was discovered to be correct. Mitigation wise, there was not much we could do about this. Since the department was unlikely to provision special resources just for us, it was down to us to find ways to reduce the impact of this. Using the different nodes and smaller tasks led to the mentioned issues, which led to arguably lower quality results. This problem could have been avoided by spreading out tests over a longer period of time.

9.5.2 Scope Creep (Moderate)

Scope creep was not a major issue in this project, arguably there were more issues with the opposite: a lack of collective understanding with regards to the general direction of the project. As discussed, this project was left open-ended on purpose to stop previous locked-in decisions preventing us from pursuing interesting work. Our goals were broad. This had benefits and drawbacks. The benefits were as expected, but there were some minor, yet key drawbacks. Lacking hard goals at the specification phase meant having to come up with our own goals. We struggled with this early on. This was solved by asking our supervisors for some more direction, which they provided the list of potential goals. This gave some much-needed direction to the project

9.5.3 Internal Conflict (Moderate)

Major internal conflict was not a factor in this project. Everyone in this project worked professionally with each other. Interpersonal issues did not occur. Even if internal conflict were to have occurred, it is unlikely that it would have impacted the project in any way; everyone was working towards the same goal, and would have been able to set aside their personal issues for the good of the collective. This positive working environment was encouraged by the flat project management strategy. Everyone was heard and there was no social pressure against saying what one believed needed saying. A voting system was established, using a simple majority system whenever someone felt that a group decision needed making. This was used on the rare occasion that there was a conflict in decision making, and in every case results were accepted without issues. This system worked well.

9.5.4 Dependency on External Libraries (Low)

The project depended on several major Python libraries. Our main fear going into the project regarding this was not being able to access those libraries. Doing so would have crippled the project. It is not unreasonable to expect that writing a library with the same capabilities as just PyTorch would take longer than the entire project. The chance of this happening was exceptionally low however. For us to lose access to these libraries would have meant the libraries were unavailable on

a wide-scale. In such a case, there would have been a strong incentive to fix it. Because of this, we were not expecting to have issues. If it did happen, we had local versions of the libraries that we could have used. Since these libraries are typically open source with an open license (33), we were within our rights to use locally downloaded versions which we had, making this problem a complete non-issue.

A different, but relevant issue was Python's loose version system. There were several times that tests would fail because the wrong version of a given package, or (in earlier tests) because the Python version itself was wrong. This was especially bad with the PyTorch module, since that module requires different versions for different hardware. This was fixed by standardising the versions across tests through adding the pip install commands to the tests. This meant that the correct version would always be installed if it did not already exist on the system. Unfortunately, this problem tends to appear when dealing with Python, and was something that we had to work around, which we successfully did.

Another issue regarding external libraries was having to learn how to use them in the first place. Most libraries we used are industry or academic standard (33) (35). This meant a lot of tutorials, and typically good documentation. They were also libraries that most members of the team had already used. As expected, there was a small learning curve at the start of the project as everyone got used to the relevant code. On rare occasions, bugs and issues with the libraries would cause tests to fail. While it was frustrating to have these unexpected failures, workarounds were usually easy to find online. When a workaround was not available, minor tweaks to our code were necessary to have it work. This never became a major blocking issue.

9.5.5 Lose Access to Codebase (Moderate)

One of the most significant risks the project faced, especially as the project progressed, was losing access to the codebase. The most obvious way this could have occurred would be through every copy of it being deleted somehow. This issue gets reduced as more copies of the code exist and was entirely mitigated by our choice to use git and GitHub for version management and online backups. This formed our decision to consider this risk "Moderate": Despite being potentially catastrophic, it was extremely unlikely to happen as long as we kept up with syncing our work

with the remote repo. Doing this also meant adhering to the 3-2-1 backup strategy commonly used in industry, that being 3 copies of the code, in 2 places, 1 of which was off-site. In our case it was closer to 6-6-1: 5 local copies on personal devices and 1 remote repo. Through frequent synchronising, we ensured that we never lost any work done, and even if we had it would have been a small loss, that would have been trivial to re-implement.

9.5.6 Group Member Leaves (Moderate)

This project was started with group of 6 people, and ended with a group of 4. The first person left before the project started in earnest; it was easy to work around their absence since they left before the plan was even written. The second group member left very late into the project, as the final testing and report writing was due. They left on very short notice, giving no hint or warning that they could not complete the work before leaving. The abruptness of their departure worsened the impact on the project. Given more time, we might have been able to better prepare for their loss. This left us in a difficult position for 3 main reasons: They were no longer available to put in work on the project, we lost access to their compute resources, and we lost the ability to ask them about the work they had done on the project.

Having less manpower was undoubtedly a huge issue; as expected, the final testing phase revealed gaps and issues in the work done. Having an extra pair of hands dedicated to evaluating and fixing these would have lightened the workload on everyone else. Having 25% extra labour would have meant being able to handle a bigger workload in this final, critical section of the project. The final sprint had a lot of different tasks that were done in parallel: Coming up with and implementing specific tests, running tests, bug-fixing, implementing and modifying existing MU and CL methods. It is not unreasonable to expect the majority of their time could have been spent working on the project. Since the majority of the report was planned to be written after they left, it also reduced the amount of time we had to spend on this final sprint, since writing said report was expected to take longer. We were forced to actively triage our tasks, and be more aggressive in cutting certain tasks from the final sprint.

Compute resources were significant issue in the final parts of the project. The

DCS compute system scheduler means having 5 people submitting a single job each will be finished before the equivalent work being submitted by a single person with 5 jobs. Losing out on an extra person to run these tests, and analyse the data on them was a significant loss for this reason. This could have been mitigated by running the tests over a longer period of time. This couldn't be factored in however as there was no warning before they left.

Losing the ability to discuss their work with them was a less significant issue, but still a significant one. An often-used method within the team was to meet with someone to discuss the code they had written before attempting to factor it into your own code. Methods like this are essential when working with Python; loose, dynamic typing can make understanding code extremely difficult without asking the person who wrote it. Since he did not make himself available after-the-fact, we were left trying to use their written code without fully understanding the structure and intent. This was not an impossible task - the code was not poorly written. It was an inconvenient and frustrating obstacle to overcome at a time where such obstacles had the chance to impact the project the most. There was also the risk of losing access to the knowledge that allowed them to write that code in the first place. Fortunately, the mitigations laid out in the specification avoided this. Throughout the project we tried to ensure that one person was never the only person to understand a specific topic. Their main tasks on the project by the point they had left were related to metrics for MU and CL quality. In this case, we all had a basic understanding of the metrics through informal conversations and stand-ups; a mitigation strategy set up in case of a situation like this. This mitigation prevented any catastrophic issues within the project, as it might have if they were the only person with an understanding of some key part of the project.

9.5.7 Not enough storage for datasets and models (Low)

Despite some thought given to running out of storage on the DCS machines, this never became an issue. The majority of the project uses the CIFAR-10 and CIFAR-100 datasets (27). These dataset are relatively small. Thought was given to potentially using larger datasets such as some subset of ImageNet (36). This would have certainly needed more space than is provided by the DCS, in which case we would have had to request more storage as laid out in the specification. When working

on the project, we never got to the point where we were held back by our dataset, hence never moving to a different, larger one.

9.6 Plan Retrospective

The original specification laid out 3 phases of the project:

1. **Setup:** Gain the resources and knowledge required to start the project. Planned to occur over the first half of Term 1
2. **Explore:** Start implementing and experimenting with existing algorithms, seeing how they interact with each other. Planned to occur over the second half of Term 1, and the first weeks of Term 2.
3. **Refine:** Look into creating new algorithms that make use of the combination of unlearning and continual learning algorithms. Planned to occur from the beginning of Term 2 until the deadline.

This schedule was roughly adhered to, but with some major bleed through regarding work done in each phase. Rather than completing one phase before beginning the next, each phase persisted beyond the planned window, continuing to a lesser degree while later work was done. That is to say that even while doing exploratory work, there was work being done regarding research, and exploration and refining were constantly running in parallel.

9.7 Meeting Schedules

The meeting schedule, and arguably the entire project, can be split into 3 periods of time: Term 1, the first half of Term 2, and the back half of Term 2 onwards. These roughly correlate with the 3 phases discussed above.

9.7.1 Term 1

In Term 1, we adhered to a rigid meeting structure - meeting once a week with our supervisors for an hour at a time, with meetings occasionally being skipped when progress was slow. This had several benefits. It meant having regular contact with our supervisors, who are individually experts in Machine Unlearning and Continual Learning. It meant they had a good understanding of where we were at, and

could steer us in the right direction. We could also take advantage of their subject knowledge. Having regular meetings like this also incentivised constant work, which stopped us from letting this work (with its far off deadline), become de-prioritised against other modules. These meetings were held in-person, which made it easier to have quick conversations about smaller parts of the project, and helped reduce the amount of miscommunications when discussing the project.

These meetings were focused on the broad strokes of the project, and building the baseline knowledge and understanding about existing Continual Learning and Machine Unlearning research and algorithms. Outside of these more formal meetings with our supervisors, we would keep in touch via Discord and email. As previously mentioned, this was useful for sharing results and getting an extra set of eyes on a problem.

This time was mostly spent doing the "Setup" and "Explore" stages of our project, as laid out in our original specification. This was spent learning and implementing existing work. We were provided a set of papers by our supervisors (1) (4) (5) (21) (24) (37), and worked our way through them, attempting to emulate some of their results through our own work. This literary analysis served to provide a strong baseline for the project. The work done to copy existing work was fairly successful, and left us with a strong understanding of the basic concepts, and the necessary questions we would need to ask.

To increase the speed with which the team collectively understood the subject, a decision was made to split the work between two teams: one focusing on the MU side, and the other the CL. The idea being that coming together after the fact to teach each other would be faster than having everyone struggling to understand at the same time. The CL team had a particular focus on the GEM and AGEM algorithms, while the MU team focused on NegGrad+, SalUn, and RUM algorithms. These algorithms were implemented, and some basic experiments copying the existing work was performed. The results of these were presented to our supervisors and second markers, along with some ideas for future work. This suggestion for future work included the algorithm that would eventually become NegGem, our first major contribution to the subject of MU and CL.

Breaking the work up like this had the advantages we expected: it sped up the amount of existing work that we could look at, and let us get some results to show

off quickly. It also had downsides however, specifically by splitting up the work like this, there was also a problem with a lack of cohesive vision as to what the project was. Since people were focused on individual halves, there was not much thought given to merging the two. This lack of vision led to a lack of velocity, by the end of Term 1, there were only 2 contributors on the group GitHub repository. This is not to say that there was no work being done by the other team members, but instead they were doing their own work disconnected from the rest of the rest of the team.

To resolve this issue, a discussion was had with the project supervisors. They sent a list of potential goals for the project. The contents of this document is attached as Appendix **Supervisor Email (CL + MUL Ideas to explore)**. These goals provided a framework for us to work with and significantly mitigated the lack of general clarity in direction.

9.7.2 Week 10 Presentation

At the end of term 1, the group gave a presentation to the supervisors and second assessor. It contained a summary of work done, and expectations for future work. By this point of the project, the majority of the implemented work was about CL and MU as separate techniques. There was a summary of our understanding around the topic, with particular focus on algorithms and methods suggested by the supervisors at the start of the project. Since the second assessor was there, it was important to lay a strong foundation for the project. The motivations, processes, definitions, and problems behind MU and CL were established, as well as the motivation behind the project as a whole. After summarising the existing literature, work done mimicking existing results from said literature was demonstrated. Of note was a demonstration of the differences between GEM and AGEM against naïve training on a test of 20 tasks. This served to demonstrate a strong baseline from which we would be able to work off for future work combining the two fields.

After setting the foundations, discussion turned to future work. At this point in time, the most promising thought was combining NegGrad+ and GEM into NegGem, which we would go on to do. There was also thought given to the final goals of the project. There were 3 possible long-term outcomes identified:

1. We find no meaningful relationship between the two fields.

2. We find some meaningful relationship between the two fields that can be utilised for improvement

3. We design our own algorithm or improve upon an existing algorithm in one or both of the fields using our findings.

After the presentation, there was time for questions. Special interest was taken with respect to SalUn, which had been brought up as part of the existing literature. Therefore, it was decided that the implementation of SalUn should be a key goal in the future.

9.7.3 The First Half of Term 2

After the Christmas break, we stopped having regular meetings with our supervisors. This was not a conscious choice; the previously scheduled meetings stopped after term 1, and everyone within the team was focusing on other work. We got out of the habit of talking with our supervisors. This meant the pressure to have constant work was not there to the same degree. This inevitably led to a de-prioritisation of the project. All things considered, it is likely that less work would have been done in this period regardless, as all the members of the team were doing similar modules, and as such had similar deadlines. The slowdown in project velocity was still notable, and could have been mitigated through better communication.

That is not to say that no work was done during this period, members of the group, especially the members of the original CL team, were still having consistent meetings regarding work in the document sent by our supervisors. This work broadly fell under the "Explore" phase laid out in the specification, with some work put towards the "Refine" phase. This was technically ahead of our schedule, but was not satisfactory for a couple of reasons. The exploring done was limited to basic algorithms, and the refining was lacking a solid baseline to work off of. It was made clear that more work had been expected by this point. The velocity with regards to the project was not 0, but it was slower than in the first term. Another, arguably bigger problem during this time is that the group got out of the habit of communicating with our supervisors. This made the issues regarding lack of work done even worse; if we had remained in constant communication the slower pace would have been noticed earlier.

9.7.4 The Mid-Term Meeting

At the midpoint of the term, a meeting was held with our supervisors to demonstrate and receive feedback on the work done since the project presentation at the end of term 1. We showed off the work we had done during this period. The minutes for this meeting can be seen in Appendix: **Mid-term Progress Meeting Minutes**.

The first thing presented was some preliminary work on using NegGrad+ with a subset of the data, inspired by memory buffers used by CL. The work suggested that there was a drop-off in unlearning quality when working with less of either dataset, but that more work needed to be done to establish proper results. A comment was made about the way in which the data was presented. Specifically, the graph lacked good labels and a title.

After, some preliminary results were shown for NegGEM: an experiment going through learning and unlearning steps. The test involved taking an untrained model, and continually learning 20 tasks using the CIFAR-100 dataset, then forgetting the last 20 tasks. This had expected the expected results. This demonstrated that the algorithm was feasible, and the intuition regarding it working was correct, albeit the specific consequences of doing so were not fully elaborated on.

After this was a demonstration of one of the key focus points for the project: that unlearning through learning was possible. Through rejecting the forget set from the GEM memory buffer, the CL process indeed led to a useful amount of MU. This was a major, albeit expected finding, and opened the door to some of the more significant research further on.

After this, there was a demonstration regarding some work done into memorisation when working with CL. Missing some of the major aspects of the topic, the results were not what we expected. Specifically, we saw that there were on average better results when using a random selection of samples instead of the most or least memorised samples. After discussing this with the supervisors, it was made clear that there were some key points that had been missed in this section of the work. Specifically we were using a poor estimator of memorisation instead of the better ones recommended in some of the existing literature (21). This was ultimately a failure of communication. If there had been stronger communication with the supervisors before the meeting, they would have been able to point out these issues

and better results could have been presented.

Finally, there was a discussion of future work. Of note was the future use of CIFAR-100, the use of a different base model over ResNet-18, future concerns about hyperparameter tuning, more testing setups, the use of weight saliency, and creating our own metrics.

CIFAR-100 was an obvious thing to look into. There is only so much to be gained with only 10, unconnected classes. CIFAR-100 opened the door to larger tests with more tasks, and the ability to compare the effects of unlearning on the coarse and fine labels. The use of another base model over ResNet-18 had a similar motivation, but was ultimately dismissed as being outside of the scope of this project, so long as we never ran into limitations with ResNet-18.

Hyperparameter tuning and more testing setups fell under the same umbrella of performing more testing. More data would lead to more avenues of research, and improve the general robustness of the project. Experimenting with different hyperparameters could give us an upper bound for the quality of our work, and provide an insight into how different parts of the algorithm interact with each other. More testing setups was also obvious: an unlearning step followed by a continual learning step might leave us in a different state to the converse. More ways of combining the two would provide more data, which was generally considered to be beneficial.

Weight saliency (5) was interesting to us. It served as a way of augmenting existing algorithms instead of being its own algorithm. We expected it to be simple to implement, for very little drawbacks and the potential for a lot of gain.

Creating our own metrics was a necessity for this project. While CL and MU had their own metrics such as tug-of-war, there was no overall metric that quantified the interplay between the two. This had become an increasing issue up to this point, and something we had collectively decided needed a solution.

The reception to this was mixed, with the supervisors critiquing the amount of work done, especially on the MU side of the project. But engaging with some of the more novel work regarding NegGem and getting MU through CL. It was during this that the expectation of more work done was established. This obviously led to a major discussion within the group. While the project had produced some interesting results, there were clear issues. The meeting caused us to reflect on

the issues with the project management and velocity, resolving to do better. We identified two main issues: Lack of communication, and a lack of focus.

Lack of communication was an obvious issue: we had become lax with communicating with our supervisors and as such, they had not been able to alert them to issues they saw with the project. Beyond that, we were communicating less with each other. This was partially because of the slower velocity; if you have nothing to talk about, it would seem reasonable to say nothing. However the lack of work led to a vicious cycle since there was less social pressure to keep up with the work. The slowdown in work was not equally distributed across all teams. The CL focused team were more active during this time when compared to the MU focused team. The issues with a slower overall velocity are obvious: less work means fewer results which means having to do more work in the future.

The lack of focus was a harder issue to pinpoint. Our issues with broad scope of this project have been laid out. There was no hard line to be aiming for, and as such we did not know how far off we were from our expectations. This was resolved by taking the list of potential objectives laid out by our supervisors, and treating them as our primary goals.

9.7.5 The Second Half of Term 2 Onwards

During this time the majority of the novel work got implemented, tested, and finished. Coming out of the meeting with the supervisors, there was a collective understanding that there needed to be stronger communication, and a stronger focus on the work laid out by our supervisors. To ensure working on the project was as easy as possible for each individual member, there was a mass refactoring of the existing code; there had been a significant amount of code sprawl with the previous code, with different people working with completely different styles. This had worked while the codebase was small, and results were being prioritised. Over time however friction was caused by the lack of any decided-on structure. The previous code was moved to a dedicated directory to ensure nothing had been lost. This, combined with the new collective understanding regarding the main goals of the project meant the general velocity increased dramatically.

To manage this increase in velocity the decision was made to have weekly meetings regarding the progress of the project within the group. These meetings were

typically held on Discord for convenience. This worked well. By this point, it was possible to carry out the bulk of the final testing. Since tests could take beyond 12 hours, an efficient workflow of doing additional work while tests were running was established. Because of these large tests, the project started to run into problems with the DCS batch compute system. These problems are discussed above.

Going into the final month of the project, the project had a significant issue: One of the group members dropped out of the university for personal reasons and would therefore be unavailable to work on the project. The specific issues regarding this are expanded in the Risks Management section above, but a critical issue was the sudden loss of manpower. This forced us to restructure the way we worked. It was extremely important that everything was working efficiently to make up for the expected work the now-absent member would have been doing. The weekly meetings were replaced with meetings every two days to keep up with all the work being done. It was during this time that the majority of the report was being written. This worked to our advantage: the workflow became discussing the results of a given test while the next test was running. Tests could be run overnight, giving most of the day to discuss its implications.

10 Legal, Social, Ethical and Professional Issues

10.1 Legal Issues

We ensure that all datasets used are open-source and properly licensed to avoid legal violations.

10.2 Social Issues

While our project primarily focuses on image classification, which typically avoids direct social implications like biased decision-making, there are still minor concerns. For instance, the selection of images in datasets could unintentionally reflect limited perspectives or representations. To mitigate this, we used industry-standard datasets, which are designed to be neutral and widely applicable for research purposes, minimising the likelihood of introducing harmful biases.

10.3 Ethical Issues

Our project could raise ethical concerns regarding the potential misuse of the models, especially since we plan to open-source our research and code. Once the code is publicly available, we no longer have control or responsibility over how the models are applied, which could lead to unintended or harmful uses.

10.4 Professional Issues

A significant professional issue in our project is the reliance on the work of other researchers. We are using a wide range of models and algorithms that have been developed and open-sourced by the machine learning research community. Given the collaborative nature of this research, we properly credit and cite all original sources of these models and algorithms.

Additionally, as we used shared compute resources provided by the Department of Computer Science, we had a professional obligation to use these resources efficiently. This included ensuring that we did not leave experiments running unnecessarily, as wasted computational time equates to unnecessary financial costs.

11 Project Evaluation

Overall, this project has been a success. We achieved our core objectives of exploring the interplay between Continual Learning and Machine Unlearning, making meaningful contributions to advancing this research area. Most notably, we developed a novel approach, NegGEM, with results demonstrating that information retained during learning can be effectively leveraged to enhance unlearning. This supports our initial hypothesis that learning and unlearning are not opposing processes but can be unified through shared mechanisms.

In addition to establishing this foundational insight, we developed multiple variations of our unified approach — including Unlearning-AGEM, Random Labelling GEM/AGEM, GEM+, and ALT-NegGEM. These variants enabled a broader explo-

ration of the trade-offs between stability, plasticity, forgetting quality, and computational efficiency, providing valuable insights for the future development of learning–unlearning algorithms.

Beyond the technical outcomes, our early project management decisions allowed us to keep our options open, maintaining flexibility to pursue promising directions as they emerged. Sharing of knowledge within the team ensured that individual developments were integrated into the broader project, enhancing our pace and breadth of exploration.

Finally, in the spirit of supporting further research in the field, we intend to open-source both our codebase and our findings. By doing so, we hope to enable future researchers to build upon our work, refine these ideas, and contribute to the broader understanding of continual learning and unlearning.

Overall, the project not only achieved its intended objectives but also laid a strong foundation for continued research into unified learning–unlearning algorithms.

12 Limitations and Future Work

While this project demonstrated the effectiveness of the NegGEM framework and its variants, it also highlighted several open questions and opportunities for further exploration.

Although we conducted extensive empirical tests, more time could have been allocated to fully compute and analyse CL-ToW scores. Currently we can only derive the metric in certain task sequences such as our [A] testing suite due to computational constraints. CL-ToW is the best metric for comparing our continual unlearning algorithms to a perfectly retrained model. Being able to compute CL-ToW scores across a wider range of scenarios would offer a much deeper understanding of the performance trade-offs involved. If we had access to greater computational resources and more time, we would have expanded this evaluation to cover different task orders and forgetting configurations.

While NegGEM and its variants showed strong performance, the broader space of unlearning algorithms remains relatively under explored. During the initial research and experimentation phase, we implemented SCRUB; however, due to time

constraints, we were unable to integrate it fully into our continual learning–machine unlearning (CL-MU) pipeline. Future work could investigate combining multiple unlearning strategies within a continual learning framework, assessing their compatibility and effectiveness across different scenarios.

Beyond unlearning algorithms, there is also substantial scope for exploring alternative continual learning methods. Our experiments focused on GEM and AGEM, which both rely on episodic memory buffers and gradient projection mechanisms. Future investigations could examine regularisation-based methods or dynamically expanding architectures, to assess whether NegGEM-style unlearning techniques generalise effectively across different types of continual learners.

A particularly interesting direction would be to develop a meta-algorithm capable of dynamically selecting unlearning strategies based on the model's training history. As continual learning progresses and more tasks are learned, certain unlearning methods may perform better at early stages, while others could be more effective after longer periods of training. A meta-controller that monitors certain metrics could adaptively choose between unlearning mechanisms over time, helping maintain best performance across the model's lifespan.

Related to this idea of dynamic adaptation, saliency mapping techniques could be leveraged not only during unlearning, but also proactively during the learning process itself. In our current framework, we apply SalUn during the forgetting process. However, applying saliency masks to gradients before the projection step during regular learning updates could reduce constraint violations. Monitoring for constraint conflicts and selectively applying saliency-based adjustments when needed could further enhance Continual Learning robustness, while also laying a stronger foundation for effective unlearning later.

Another promising avenue involves exploring generative replay to enhance memory buffer privacy. In this project, Continual Learning and Unlearning were based on carefully selected real data subsets, but these examples proved particularly vulnerable to membership inference attacks. If generative models, such as GANs or diffusion models, could synthesise proxy examples that retain learning-relevant features while obfuscating membership, it might be possible to preserve both model performance and data privacy. Future work could explore lightweight generative replay architectures, analysing the trade-offs between sample quality, memory size,

and unlearning effectiveness.

Finally, while our experiments focused on relatively small datasets to enable rapid experimentation and benchmarking, scaling up to larger and more complex datasets remains an important future step. Evaluating NegGEM and related frameworks on datasets such as ImageNet, or under real-world continual learning scenarios involving heterogeneous tasks and longer training horizons, would provide a more rigorous test of their practical viability. Issues such as memory scalability, computational overhead, and unlearning latency would become critical factors at scale, and addressing them is essential for real-world deployment.

In summary, although this project developed a strong foundation for integrating continual learning and machine unlearning, there remains significant scope to extend, generalise, and stress-test the proposed methods. We hope that the challenges raised and directions outlined here will inspire further research toward building scalable, adaptable, and certifiable continual unlearning systems.

References

[1] L. Wang, X. Zhang, H. Su, and J. Zhu, "A comprehensive survey of continual learning: Theory, method and application," 2024. [Online]. Available: https://arxiv.org/abs/2302.00487

[2] EU Parliament, "General data protection regulation," April 2016, article 17: Right to erasure ('right to be forgotten'). [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679

[3] EDPB, "Opinion 28/2024 on certain data protection aspects related to the processing of personal data in the context of ai models," European Data Protection Board (EDPB), EDPB Opinion, December 2024. [Online]. Available: https://www.edpb.europa.eu/system/files/2024-12/edpb_opinion_202428_ai-models_en.pdf

[4] M. Kurmanji, P. Triantafillou, J. Hayes, and E. Triantafillou, "Towards unbounded machine unlearning," 2023. [Online]. Available: https://arxiv.org/abs/2302.09880

[5] C. Fan, J. Liu, Y. Zhang, E. Wong, D. Wei, and S. Liu, "Salun: Empowering machine unlearning via gradient-based weight saliency in both image classification and generation," 2024. [Online]. Available: https://arxiv.org/abs/2310.12508

[6] D. Lopez-Paz and M. Ranzato, "Gradient episodic memory for continuum learning," *CoRR*, vol. abs/1706.08840, 2017. [Online]. Available: http://arxiv.org/abs/1706.08840

[7] EU Parliament, "General data protection regulation," April 2016. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679

[8] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[9] Z. Li and D. Hoiem, "Learning without forgetting," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 12, pp. 2935–2947, 2017.

[10] A. Ayub and A. R. Wagner, "Eec: Learning to encode and regenerate images for continual learning," *arXiv preprint arXiv:2101.04904*, 2021.

[11] J. Gallardo, T. L. Hayes, and C. Kanan, "Self-supervised training enhances online continual learning," *arXiv preprint arXiv:2103.14010*, 2021.

[12] S. V. Mehta, D. Patil, S. Chandar, and E. Strubell, "An empirical investigation of the role of pre-training in lifelong learning," *Journal of Machine Learning Research*, vol. 24, no. 214, pp. 1–50, 2023.

[13] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," *arXiv preprint arXiv:1312.6211*, 2013, section 4 analyses how task dissimilarity amplifies forgetting. [Online]. Available: https://arxiv.org/abs/1312.6211

[14] M. D. Lange, G. van de Ven, and T. Tuytelaars, "Continual evaluation for lifelong learning: Identifying the stability gap," in *Proceedings of the*

International Conference on Learning Representations (ICLR), 2023, spotlight paper; shows that performance drops grow as task similarity decreases. [Online]. Available: https://arxiv.org/abs/2205.13452

[15] A. Chaudhry, M. Ranzato, M. Rohrbach, and M. Elhoseiny, "Efficient lifelong learning with a-gem," 2019. [Online]. Available: https://arxiv.org/abs/1812.00420

[16] M. Puderbaugh and D. Emmady, P. *Neuroplasticity.* Treasure Island (FL): StatPearls Publishing, 2025, updated 2023-05-01. [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK557811/

[17] D. Lopez-Paz and M. Ranzato, "Gradient episodic memory for continual learning," *Advances in neural information processing systems*, vol. 30, 2017.

[18] Y. Cao and J. Yang, "Towards making systems forget with machine unlearning," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 463–480.

[19] A. Fabbrizzi *et al.*, "Policy advice and best practices on bias and fairness in ai," *Ethics and Information Technology*, vol. 24, pp. 1–20, 2022. [Online]. Available: https://link.springer.com/article/10.1007/s10676-024-09746-w

[20] C. G. Northcutt, A. Athalye, and J. Mueller, "Pervasive label errors in test sets destabilize machine learning benchmarks," in *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2021. [Online]. Available: https://arxiv.org/abs/2103.14749

[21] K. Zhao, M. Kurmanji, G.-O. Bărbulescu, E. Triantafillou, and P. Triantafillou, "What makes unlearning hard and what to do about it," 2024. [Online]. Available: https://arxiv.org/abs/2406.01257

[22] N. Carlini, S. Chien, M. Nasr, S. Song, A. Terzis, and F. Tramer, "Membership Inference Attacks From First Principles," *arXiv*, Dec. 2021.

[23] J. Hayes, I. Shumailov, E. Triantafillou, A. Khalifa, and N. Papernot, "Inexact Unlearning Needs More Careful Evaluations to Avoid a False Sense of Privacy," *arXiv*, Mar. 2024.

[24] A. Golatkar, A. Achille, and S. Soatto, "Eternal sunshine of the spotless net: Selective forgetting in deep networks." *Proceedings of the IEEE/CCVF Conference on Computer Vision and Pattern Recognition*, pp. 9304–9312, 2020.

[25] V. Feldman, "Does learning require memorization? a short tale about a long tail," *Proceedings of the 52nd Annual ACM SIGACTC Symposium on Theory of Computing*, pp. 954–959, 2020.

[26] M. Goldblum, S. Reich, L. Fowl, R. Ni, V. Cherepanova, and T. Goldstein, "Unraveling Meta-Learning: Understanding Feature Representations for Few-Shot Tasks," *arXiv*, Feb. 2020.

[27] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:18268744

[28] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert, "icarl: Incremental classifier and representation learning," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2017, pp. 2001–2010.

[29] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[30] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*. Cambridge University Press, 2023, https://D2L.ai.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[32] Warwick batch system. https://warwick.ac.uk/fac/sci/dcs/intranet/user_guide/batch_compute/. [Online]. Available: https://warwick.ac.uk/fac/sci/dcs/intranet/user_guide/batch_compute/

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[34] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance mixed precision," in

2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, May 2018. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2018.00091

[35] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter notebooks–a publishing format for reproducible computational workflows," in *Positioning and power in academic publishing: Players, agents and agendas*. IOS press, 2016, pp. 87–90.

[36] ImageNet, "About imagenet," 2020, accessed on 23.10.2024. [Online]. Available: https://image-net.org/about.php

[37] D.-W. Zhou, H.-L. Sun, J. Ning, H.-J. Ye, and D.-C. Zhan, "Continual learning with pre-trained models: A survey," in *IJCAI*, 2024, pp. 8363–8371.

A User Guide

A.1 Installation

The codebase developed for this project is available at the following GitHub repository: `github.com/DROP-TABLE-CS407/Machine-Unlearning-x-Continual-Learning`.

To reproduce experiments on the University of Warwick DCS Batch Compute System, no additional configuration is required beyond standard batch compute access. All dependencies and paths are pre-configured to run out-of-the-box in that environment. For local deployment, users must modify certain directory paths within the main execution script: `./negGemGradSalun.py`
These edits are necessary to ensure correct file access and dataset loading when running outside the DCS infrastructure.

A.2 Running an Experiment

Experiments can be launched using the provided `sbatch` scripts for batch scheduling, or directly via terminal using `python3.12`. Sample job scripts are included and can be adapted with minimal changes. Each run performs continual learning followed by selective unlearning, logging both accuracy and privacy metrics throughout.

Example Command

```
time python3.12 negGemGradSalun.py \
  --algorithm neggem \
  --alpha 0.9 \
  --number_of_gpus 3 \
  --learn_mem_strength 0.5 \
  --learn_batch_size 10 \
  --unlearn_mem_strength 0.6 \
  --unlearn_batch_size 10 \
  --average_over_n_runs 3 \
  --salun 1 \
  --salun_strength 0.2 \
```

```
--mem_learning_buffer 1 \
--learning_buffer_split 0.2 \
--learning_buffer_type least \
--mem_unlearning_buffer 1 \
--unlearning_buffer_split 0.2 \
--unlearning_buffer_type most
```

Argument Descriptions

Argument	Description
`--algorithm`	Unlearning method: `neggem`, `negagem`, `RL-GEM`, `RL-AGEM`, `ALT-NEGGEM`, `neggrad`
`--alpha`	Projection factor used by `neggrad`-style algorithms
`--number_of_gpus`	Number of GPUs to allocate for parallel execution
`--learn_mem_strength`	Margin for GEM-based continual learning projections
`--learn_batch_size`	Batch size during continual learning updates
`--unlearn_mem_strength`	Projection margin during unlearning steps
`--unlearn_batch_size`	Batch size for unlearning
`--average_over_n_runs`	Number of randomised runs to average over
`--salun`	Toggle for SaLUn filtering (1 = enabled, 0 = disabled)
`--salun_strength`	Percentage of gradient elements retained via SaLUn (e.g. 0.2 = top 20%)
`--mem_learning_buffer`	Enables memorisation-based buffer selection for learning
`--learning_buffer_split`	Proportion of learning buffer based on score vs random sampling
`--learning_buffer_type`	Sample ranking: `least`, `most`, or `random`
`--mem_unlearning_buffer`	Enables memorisation-based buffer for unlearning
`--unlearning_buffer_split`	Score/random mix ratio for unlearning buffer
`--unlearning_buffer_type`	Ranking type: `most`, `least`, or `random`

Additional Notes

- All tunable hyperparameters are declared and documented in `negGem/args.py`.
- Class/task order is randomly shuffled at each run to simulate realistic continual learning conditions.
- All outputs (including plots and CSV files of performance metrics) are saved in a results directory prefixed with `Results_`.

B Supervisor Email (CL + MUL Ideas to explore)

1. Fundamentally, both MUL and CL need to deal differently with forgetting and learning.

   a. MUL needs to **remember** (**not CF**) old knowledge/tasks/data and **forget** some old knowledge/tasks/data.

   b. CL needs to **remember** (**not CF**) old knowledge/tasks/data and **learn** new knowledge/tasks/data.

   So they share this **dual goal**. This duality is dealt with differently, though.

- In GEM this is achieved using a loss on new learning with a constraint on the loss of previous knowledge learned.
- In NegGrad+ (and SCRUB) this is achieved by having different terms in the loss function (with weights treated as hyper-parameters).

So, can we see how dealing with the two ways of dealing this duality affects each other? For example, a new version, call it, **NegGrad++** can use a constraint on the loss of the retain set, a la GEM, versus a loss term for the retain set? And vice versa, in GEM (call it **GEM+** or **GEM-** or **GEM+-**) having perhaps an explicit loss term for avoiding CF instead of having these explicit constraints to satisfy? This may affect running times and accuracy... But how?

Intuition tells me that this will make a key difference. For example, in RUM we found that using a metric like ToW (or ToW-MIA) showed that unlearning low-memorized examples is easier etc. Using constraints on the loss of old tasks/data

(ie "retain data/tasks" in the MUL lingo) bypasses such difficulties (meeting the constraints in loss of old tasks deals with this implicitly).... So, very eager to see how this would work...

2. MUL algos like NegGrad+ need to operate on the retain and forget sets. In CL, we have explicit subsets of these in the replay buffers. Can we apply NegGrad+ using just these buffers instead of the whole of retain set? This would be interesting:

    a. first, it will be faster as we will be operating only on subsets.

    b. Secondly, we can see whether just forgetting a subset can lead to **generalized forgetting** (ie can we forget a task by explicitly forgetting only on a subset of the task?)

Or in GEM-like CL, we can apply NegGrad+ on the gradients of learned tasks in the replay buffers as we don't have the actual data examples in the buffers).

3. "MUL through the lens of CL":

Treat an unlearning request/step as just another new task in CL. Instead of running an MUL algo, run a CL algo for the forget task as well. For instance, GEM's loss is traditionally playing with gradient descend on the new data/task while respecting loss bounds on previous tasks. In our new version, the new algo, "**UnGEM"** will be doing

- a gradient **ascend** on the task to forget
- s**ubject to the constraint that the loss on the other tasks not decreasing!**

4. [Paris help here]: I am not sure if people have looked at how to select the examples/gradients to put in the replay buffers?

There is increasing interest in the research community as to how memorization affects MUL (and previously learning in general).

Do people know how memorization affects the contents in replay buffers within CL? Specifically, how are/should the contents of the replay buffers be affected by the different memorization degrees of training examples?

Assume we can easily/efficiently compute the memorization scores fo different examples (a la our Nneurips and fitml papers)...

5. This paper `https://arxiv.org/abs/2202.00155` presents 'fortuitous forgetting' basically intervening during training to make the network forget some info while enhance the key information, which avoids overfitting etc. So the paper looks at successive iterations within a single training step. I think there is room here to ride on / adapt their observations and algos (eg for targeted forgetting) in subsequent 'iterations' within a CL setting... to make subsequent learning and unlearning easier...

C Mid-term Progress Meeting Minutes

Date/Time: Friday, 28th February 2025, 12:00 – 13:00

Location: Microsoft Teams

Organizer: Krishi Mistry

Note-taker: Thomas Marsh

Participants:

- **Students:** Guney Coban, Paul Joakim, Krishi Mistry, Thomas Marsh, Gerald Ramos
- **Supervisors:** Paris Giampouras, Peter Triantafillou

Agenda and Discussion Summary:

1. **Subset NegGrad**
    - Preliminary work using NegGrad+ with a subset of the dataset was presented, inspired by CL memory buffers.
    - Observed drop-off in unlearning quality when subsets were reduced.
    - Supervisors noted the graph presentation lacked good labels and a title.
    - Further work needed to establish proper results.
2. **NegGem Demonstration**

- Preliminary results for NegGem were shown, involving sequential learning and forgetting of tasks on CIFAR-100.
- Performance dropped as expected for the final 10 tasks.
- Demonstrated that learning and unlearning steps could be alternated, though specific consequences remain unexplored.

3. **Learning via Unlearning**

   - Demonstrated that rejecting forget-set samples from GEM memory during CL produced MU.
   - This confirmed a key project hypothesis and enabled further research directions.

4. **Memorisation Experiments**

   - Results showed random sample selection performed better than selecting most/least memorised samples.
   - Supervisors identified flaws: a poor estimator of memorisation was used instead of better literature-recommended methods.
   - Acknowledged as a failure of prior communication with supervisors.

5. **Future Work Discussion**

   - Continued use of CIFAR-100.
   - Evaluating alternative base models beyond ResNet18.
   - Hyperparameter tuning considerations.
   - Expansion of testing setups.
   - Application of weight saliency.
   - Development of custom evaluation metrics.

Key Takeaways:

- Presentation quality (graph labelling) needs improvement.
- Early supervisor engagement is crucial to avoid methodological errors.
- Foundations established for advancing unlearning via CL techniques.

D Code Snippets

We provide the code snippets for each of the gradient projection functions: NegGrad+, AGEM (15) and GEM (6).

```
def neggrad2plus(gradient, memories, alpha = 0.5):
    gR = memories.t().double().mean(axis=0).cuda()
    gS = gradient.contiguous().view(-1).double().cuda()
    x = gR*alpha + gS * (1-alpha)
    gradient.copy_(torch.Tensor(x).view(-1, 1))
```

Listing 1: NegGrad+ Projection Function

```
def agemprojection(gradient, gradient_memory):
    gref = gradient_memory.t().double().sum(axis=0).cuda()
    g = gradient.contiguous().view(-1).double().cuda()

    dot_prod = torch.dot(g, gref)

    dot_prod = dot_prod/(torch.dot(gref, gref))

    x = g + gref * abs(dot_prod)
    gradient.copy_(torch.Tensor(x).view(-1, 1))
```

Listing 2: AGEM Projection Function

```
def project2cone2(gradient, memories, margin=0.5, eps=1e-3):
    memories_np = memories.cpu().t().double().numpy()
    gradient_np = gradient.cpu().contiguous().view(-1).double().numpy
    ()

    t = memories_np.shape[0]
    P = np.dot(memories_np, memories_np.transpose())
    P = 0.5 * (P + P.transpose()) + np.eye(t) * eps
    q = np.dot(memories_np, gradient_np) * -1
    G = np.eye(t)
    h = np.zeros(t) + margin
    v = quadprog.solve_qp(P, q, G, h)[0]
    x = np.dot(v, memories_np) + gradient_np
    gradient.copy_(torch.Tensor(x).view(-1, 1))
```

Listing 3: GEM Projection Function

```
def basic_mia(model,
```

```
            forget_data: np.ndarray, forget_labels: np.ndarray,
            test_data:   np.ndarray, test_labels:   np.ndarray,
            task_idx: int,
            *, device=`cpu`, clip_val=400.0, cv_splits=5,
            rng: np.random.Generator | None = None) -> Dict[str,
    float]:
    ```Balanced  l o s s based  MIA using logistic regression.```
    rng = rng or np.random.default_rng()

    # Determine balanced sample size
    n = min(len(forget_data), len(test_data)) // 2
    if n < 2:
        return {`type`: `basic`, `attack_acc`: .5, `cv_acc`: .5, `auc
    `: .5, `task`: task_idx}

    # Sample 2*n from each to allow train+eval
    idx_f = rng.choice(len(forget_data), 2 * n, replace=False)
    idx_t = rng.choice(len(test_data),   2 * n, replace=False)

    f_data, t_data = forget_data[idx_f], test_data[idx_t]
    f_labels, t_labels = forget_labels[idx_f], test_labels[idx_t]

    # Split into train/eval
    f_train, f_eval = f_data[:n], f_data[n:]
    t_train, t_eval = t_data[:n], t_data[n:]
    f_lab_train, f_lab_eval = f_labels[:n], f_labels[n:]
    t_lab_train, t_lab_eval = t_labels[:n], t_labels[n:]

    # Create loaders
    loaders = {
        `f_train`: _to_loader(f_train, f_lab_train, device=device),
        `t_train`: _to_loader(t_train, t_lab_train, device=device),
        `f_eval`:  _to_loader(f_eval,  f_lab_eval,  device=device),
        `t_eval`:  _to_loader(t_eval,  t_lab_eval,  device=device)
    }

    # Collect per-sample loss
    lf_tr = _collect(model, loaders[`f_train`], task_idx, op=`loss`)
    lt_tr = _collect(model, loaders[`t_train`], task_idx, op=`loss`)
    lf_ev = _collect(model, loaders[`f_eval`],  task_idx, op=`loss`)
    lt_ev = _collect(model, loaders[`t_eval`],  task_idx, op=`loss`)
```
```

```
41
42      clip = lambda v: np.clip(v, -clip_val, clip_val)
43      lf_tr, lt_tr, lf_ev, lt_ev = map(clip, [lf_tr, lt_tr, lf_ev, lt_ev
        ])
44
45      X_train = np.concatenate([lf_tr, lt_tr]).reshape(-1, 1)
46      y_train = np.concatenate([np.ones_like(lf_tr), np.zeros_like(lt_tr
        )])
47      X_eval  = np.concatenate([lf_ev, lt_ev]).reshape(-1, 1)
48      y_eval  = np.concatenate([np.ones_like(lf_ev), np.zeros_like(lt_ev
        )])
49
50      # Manual CV
51      split = len(X_train) // cv_splits
52      cv_scores = []
53      for k in range(cv_splits):
54          s = slice(k * split, (k + 1) * split)
55          tr_idx = np.r_[0:s.start, s.stop:len(X_train)]
56          clf = LogisticRegression(max_iter=1000).fit(X_train[tr_idx],
        y_train[tr_idx])
57          cv_scores.append(accuracy_score(y_train[s], clf.predict(
        X_train[s])))
58
59      # Final attack classifier
60      attacker = LogisticRegression(max_iter=1000).fit(X_train, y_train)
61      acc = accuracy_score(y_eval, attacker.predict(X_eval))
62      auc = roc_auc_score(y_eval, attacker.predict_proba(X_eval)[:, 1])
        if len(np.unique(y_eval)) == 2 else .5
63
64      return {
65          `type`: `basic`,
66          `attack_acc`: float(acc),
67          `auc`: float(auc),
68          `task`: task_idx
69      }
```

Listing 4: Basic MIA Function

```python
from __future__ import annotations

import os
from typing import Dict, List, Set, Tuple

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ------------------------------------------------------------
#  Task curriculum (edit here only if your curriculum differs)
# ------------------------------------------------------------
TASK_SEQUENCE: List[int] = [
     0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    10,11,12,13,14,15,16,17,18,19,
    -19,-18,-17,-16,-15,-14,-13,-12,
    -11,-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,
]

def _split_sets(iter_idx: int) -> Tuple[Set[int], Set[int]]:
    """Return (S, R) after iteration *iter_idx*.

    S     forgotten tasks, R     retained tasks.
    """
    learned, forgotten = set(), set()
    for idx in range(iter_idx + 1):
        tid = TASK_SEQUENCE[idx]
        (learned if tid >= 0 else forgotten).add(abs(tid))
    return forgotten, learned - forgotten


def _mean_abs_diff(cur: pd.Series, base: pd.Series, cols: Set[int]) ->
     float:
    return 0.0 if not cols else float(np.abs(cur[list(cols)] - base[
    list(cols)]).mean())

def calculate_metrics_single(train_csv: str, test_csv: str):
    """Return per-iteration lists of
         (Tow, 1-da_s, 1-da_r, 1-da_test)."""

```

```
    if not (os.path.isfile(train_csv) and os.path.isfile(test_csv)):
        raise FileNotFoundError(f"Missing file(s): {train_csv} / {test_csv}")

    df_tr = pd.read_csv(train_csv, index_col=0)
    df_te = pd.read_csv(test_csv,  index_col=0)

    if df_tr.shape != df_te.shape:
        raise ValueError(f"Shape mismatch between {train_csv} and {test_csv}")

    task_cols = [c for c in df_tr.columns if str(c).lstrip("-").isdigit()]
    if len(task_cols) < 20:
        raise ValueError("Expected  20  task columns (0-19)")

    tow, one_m_ds, one_m_dr, one_m_test = [], [], [], []

    for i in range(len(df_tr)):
        if i <= 18:  # original-model window     unity scores
            tow.append(1.0); one_m_ds.append(1.0); one_m_dr.append(1.0); one_m_test.append(1.0)
            continue

        S, R = _split_sets(i)
        row_tr, row_te = df_tr.iloc[i], df_te.iloc[i]
        base_tr, base_te = df_tr.iloc[38 - i], df_te.iloc[38 - i]

        da_s   = _mean_abs_diff(row_tr, base_tr, S)
        da_r   = _mean_abs_diff(row_tr, base_tr, R)
        retain = [t for t in R if str(t) in row_te.index]
        da_te  = abs(row_te[retain].mean() - base_te[retain].mean()) if retain else 0.0

        one_m_ds.append(1.0 - da_s)
        one_m_dr.append(1.0 - da_r)
        one_m_test.append(1.0 - da_te)
        tow.append((1 - da_s)*(1 - da_r)*(1 - da_te))

    return tow, one_m_ds, one_m_dr, one_m_test

```

```
75
76 def calculate_all_metrics(csv_files: Dict[str, Dict[str, str]]):
77     per_iter = {}
78     for label, paths in csv_files.items():
79         tow, om_ds, om_dr, om_te = calculate_metrics_single(paths["
    train"], paths["test"])
80         per_iter[label] = {"Tow": tow, "1-ds": om_ds, "1-dr": om_dr, "
    1-dtest": om_te}
81     return per_iter
82
83 def summarise_metrics(per_iter):
84     summary = {}
85     for label, metrics in per_iter.items():
86         summary[label] = {k: float(np.mean(v[20:])) for k, v in
    metrics.items()}
87     return summary
88
89 def plot_tow(per_iter):
90     plt.figure(figsize=(12, 6))
91     for label, m in per_iter.items():
92         plt.plot(range(20, len(m["Tow"])), m["Tow"][20:], lw=2, label=
    label)
93     plt.axhline(1.0, c="k", ls="--", lw=1)
94     plt.xlabel("Iteration"); plt.ylabel("ToW score")
95     plt.title("Continual-Unlearning Tug-of-War     Eight Methods")
96     plt.ylim(0, 1); plt.grid(alpha=.3); plt.legend(); plt.tight_layout
    ()
97     plt.savefig("tow_scores_all_methods.png", dpi=600, transparent=
    True)
98     plt.show()
99
100 def main():
101     print("Computing ToW scores and averaged metrics for all runs    "
    )
102     per_iter = calculate_all_metrics(CSV_FILES)
103     summary  = summarise_metrics(per_iter)
104     plot_tow(per_iter); plot_avg_radar(summary)
105     #     exports
106     pd.DataFrame(summary).T.to_csv("avg_metrics_summary.csv",
    index_label="run")
107     for lbl, m in per_iter.items():
```

```
        f = f"ToW_scores_{lbl.replace(' ', '_').replace('+', 'plus').replace('  ','alpha')}.csv"
        pd.DataFrame({"Tow": m["Tow"]}).to_csv(f, index_label="iter"); print(f"  saved {f}")
    print("  saved avg_metrics_summary.csv, tow_avg_radar.png, tow_scores_all_methods.png")
```

Listing 5: CL-ToW Calculation Script

E CIFAR Image Examples

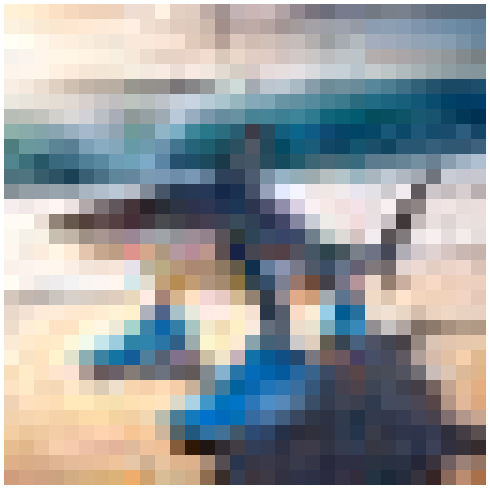


Figure 46. Example shark from CIFAR-10.

Figure 47. Example frog from CIFAR-10.