Freie Universität Berlin

Master thesis at the Institute for Computer Science at Freie Universität Berlin

Artificial Intelligence and Machine Learning Group

Analysis of the Generative Replay Algorithm and Comparison with other Continual Learning Strategies on Newly Defined Non-stationary Data Stream Scenarios

Florian Mies

First Examiner:	Prof. Dr. Eirini Ntoutsi
Second Examiner:	Prof. Dr. Daniel Göhring
Advisor:	Philip Naumann

Berlin, September 8, 2022

Abstract

Training neural networks on newly available data leads to catastrophic forgetting of previously learned information. The naive solution of retraining the neural network on the entire combined data set of old and new data is costly and slow and not always feasible when access to the old data is restricted. Various strategies have been proposed to counter catastrophic forgetting, among them Generative Replay, where together with the discriminator a second, generative model is trained to learn the distribution of the training data. When new data becomes available the generator produces data resembling the old data set and the neural networks' training is continued on the combination of the new data and the generated *replay* data. In this thesis, we implement this method and add it to the Open Source Continual Learning Library *Avalanche*. We then compare several variations of how to use Generative Replay in order to understand better when the method works best, using the common benchmark scenario splitMNIST as our testing scenario. We then argue that benchmarks like these do not necessarily correspond to real-life settings and we propose a new scenario to address this issue. We evaluate several strategies on the new scenario and find that state-of-the-art method *iCaRL* is outperformed by Generative Replay. However, we also find that Generative Replay is not easy to use and it requires knowledge on the underlying scenario to adjust it to work properly.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

September 8, 2022

Florian Mies

Contents

1 Introduction

1.1 Motivation

In recent years the Artificial Intelligence (AI) community has produced impressive results in various areas outperforming humans in individual tasks, such as playing Go and object recognition [42, 46]. This has often been achieved by training ever deeper neural networks, which are computational models composed of multiple processing layers capable of solving these tasks. The models contain an ever increasing amount of trainable parameters to such extent that the staggering 175 billion parameters of Open AI's GPT-3 model have been surpassed quite comfortably by the 530 billion parameters of NVIDIA and Microsoft's Megatron-Turing Natural Language Generation model in October 2021, making it the largest ever language model trained to date [48]. According to its original paper, training the GPT-3 model already takes "several thousand petaflop/s-days of compute" [2], which has been estimated to cost anywhere between 4-20 million USD for a single training run of the model. Where the lower bound looks at the cheapest cloud computing option, neglecting that a sophisticated orchestration of thousands of graphics processing units (GPUs) in parallel has to be performed in order to train such models. [14, 35, 47]

The sheer amount of hardware and electricity costs to train state-of-the-art models quickly raises the question how we can efficiently update a neural network once new data is available. Ideally we could update the model by teaching it new information whilst retaining its previous knowledge. Naively continuing the training phase of a model and feeding it only the new data leads to the phenomenon known as *Catastrophic Forgetting* (CF) [12], where the model quickly forgets the knowledge it acquired on previous tasks. Therefore, in practice the update process is commonly done by mixing the new data with the old data and effectively retraining the model from scratch. This approach, however, is not only costly, as we have learned above, but it is also not always feasible in cases where we do not have access to the entire previously used data set (for instance due to limited storage or for data privacy reasons) [10]. The entire research branch of *Continual Learning* (CL) was born to study this problem and is determined to find ways to mitigate catastrophic forgetting while efficiently adjusting models to previously unseen data [6].

1.2 Our Corner of the Problem

Many methods have been proposed which can be roughly grouped into three approaches: the first are regularization-based methods which penalize changing those parameters of the model that contain information on previous tasks [23, 29, 51]. The second are architectural-based methods which add neurons to the network and change its architecture in order to provide new trainable parameters when new tasks are to be learned [8, 43]. And lastly we have the group of rehearsal-based methods which describes all strategies that have access to a pool of real or synthetic data from the past which is then used to mitigate catastrophic forgetting when training on new data [16, 40, 44]. There has also been a range of strategies proposed that combine several of the approaches above [32, 37, 39]. We will present them in greater detail in chapter 3.2.2. For our research, the group of rehearsal-based approaches will be most relevant, as we will mainly focus on the Generative Replay strategy which falls into this category. From practical observations as well as from theoretical findings, it has been suggested that replay-based approaches offer the most hopeful path of finding an algorithm to solve the quest of continual learning. For example, [49] has found strong evidence that some sort of replay might be necessary to solve the CL problem on more complex data sets given that replay-based methods produced good results where all other methods failed. Furthermore, it has been mathematically shown that "optimal CL algorithms would have to solve an NP-HARD problem and perfectly memorize the past" [24], which explains why it has proven so difficult to solve the CL problem while at the same time the "results provide a theoretically grounded confirmation of recent benchmarking results, which found that CL algorithms based on experience replay, core sets and episodic memory were more reliable" [24]. In particular there is one replay-based method that we want to study and understand in greater depth as many reviews have either neglected it or merely scratched the surface. The method is called *Generative Replay* (GR) for which no past data has to be stored explicitly but instead a generative model is trained alongside the discriminator to generate as much data as needed at any given moment [44]. This method has the huge advantage of being applicable in settings where it is infeasible to store any data points, i.e. where we are in an online streaming setting. In this thesis we want to understand the method better and we are asking questions such as: what are the intricacies of implementing and applying generative replay? Which aspects have to be taken into consideration and what are its limitations? This means concretely that we will test different variants of the algorithm on existing benchmarks to determine when it performs best. In a next step we will look at new scenarios that we believe to be closer to real-world applications and evaluate Generative Replay along with other existing strategies on those scenarios.

1.3 Structure of this Thesis

The thesis will be structured as follows: After concluding the introduction with this section, we will, in Chapter 2, give an overview of the related works that our research is embedded in. Chapter 3 then contains the theoretical foundations for our research. Those include a mathematical introduction to Artificial Neural Networks (Section 3.1) and then a definition of what it means to train those in a continual learning setting (Section 3.2). Finally we will describe the Generative Replay algorithm which is used to facilitate Continual Learning in Artificial Neural Networks (Section 3.3). This algorithm is placed at the core of the remaining chapters in which we answer our research questions regarding this method. As a first contribution to the existing research we have implemented the algorithm and added it to the open-source continual learning framework Avalanche. We hope that this implementation can serve as an entry point for other researchers. We describe the Avalanche library and our contribution to it in Chapter 4. Then, in Chapter 5, we will compare various variations of the algorithm on the continual learning benchmark scenario spliMNIST regarding their performance. The variations of the algorithms are concerning exactly how we replay the generated data. We find that in order to efficiently learn incrementally, we need to either increase the number of replayed samples over time or we have to increase their importance in the loss calculation, such that new tasks a neural network learns are considered less important as the existing knowledge of the model grows. We then show that similar results can be achieved by using a conditional generator and enforcing equally distributed replay samples among classes. In Section 5.3 we rerun an experiment from the original paper ([44]) using our own implementation of the algorithm. Although we do not reproduce the exact results, we find that the algorithm is indeed robust even with regards to reinitializing the model's weights before training a new task. We conclude that the generator, which in our case uses a very simple architecture, acts as a bottleneck for the performance of the algorithm. We then shift the focus to a more scenario-centric perspective. In Chapter 6, we look at two scenarios which we believe to counter some of the unrealistic assumptions that are embedded into established benchmark scenarios. We compare Generative Replay with other methods on a new scenario where the data distribution of the data stream changes over time and classes can reappear in later experiences. We find that GR outperforms state-of-the-art method *iCaRL* on this scenario, a method we describe in Chapter 2. In another scenario, we reduce the amount of samples that are trained in each experience to a single batch. Here GR fails and some of the weaknesses of the method come to light. In particular, the complex interplay of discriminator and generator make the method difficult to handle. In comparison, we find that other methods, in particular GEM (Gradient Episodic Memory), excel in both scenarios and are easy to use. Finally we summarize and conclude our findings in Chapter 7.

2 Related Work

We want to briefly embed our work and research topic into the context of other related works and summarize the insights that have been generated so far. At the same time we also want to point out how this work is contributing something new that, to the best of our knowledge, has been underrepresented or overlooked in the related literature.

The generative replay method, which is the center piece of this analysis, was proposed in [44] and we will explain the paper's contribution in detail in Section 3.3. We noticed, however, that the algorithm is described on a high level perspective and that the paper does not describe the intricacies of implementing and applying this algorithm. We identified this as missing information in the study of this algorithm and we dedicate Chapter 5 to filling this gap. Another work that mainly focuses on Generative Replay is [49]. It evaluates the method on more complex data sets and introduces extensions of the algorithm, that are directly inspired by the biology of the mammalian brain and that increase its performance particularly on more challenging data sets, the authors call it Brain-inspired Replay. Furthermore the work establishes the robustness of Generative Replay and it shows that those methods that do not use any form of memory replay fail on the more complex data set. The range of other methods that have been proposed to counter catastrophic forgetting will be briefly summarized in Section 3.2.2. There are several works that evaluate and compare these methods in terms of performance on different benchmark data sets. [50]

has established a framework for evaluation which we will describe in 3.2.3. It also compared a range of continual learning strategies in this framework. Another such evaluation of existing methods but on more complex data sets can be found in [10]. Both surveys postulate the iCaRL strategy [39] to be state-of-the-art.
iCaRL (Incremental Classifier and Representation Learning) stores an exemplar set (a set of samples) of each class it encounters during training, with some upper bound K on the total amount of stored samples (and thus is a rehearsal-based method). It trains a convolutional neural network consisting out of a feature extractor and a classifier. The network, however, is only used for representation learning (of images) and not for their classification. When it comes to inference, iCaRL classifies an image by forwarding it through the feature extractor to obtain its low dimensional representation and then comparing it to the mean representations of each exemplar set, the class prototype, using the standard euclidean norm. The class of the nearest prototype is chosen to be the input image's predicted class. The training of the network is performed on the stored exemplar sets combined with the new data and uses both a classification and a distillation loss for optimization. With each new class, a new exemplar set is created and the existing exemplar sets are reduced in size [39].

Since many works have pursued a strategy-centric view in the sense that they develop new strategies that increase the performance on commonly used benchmarks, we want to adopt a more scenario-centric perspective. Already in other works, criticism has been raised regarding the established benchmark scenarios, saying that these do not reflect scenarios that would be encountered by real-life applications for continual learning, such as an autonomous agent learning to survive in changing environments [16, 32]. We join this argumentation and we aim to test Generative Replay in different kinds of scenarios that resemble more those of real-life applications. Both [16] and [32] introduce new strategies designed to be able to learn even with small increments of data. Generative Replay has not yet been compared to other methods under such conditions, which we see as a gap in research that we would like to fill (Section 6.3). In Section 6.2, we design a new scenario to observe how common CL strategies react to an imbalanced data set as well as disappearing and reappearing tasks over time. We believe that such a scenario has not been proposed in the literature so far and we actually find that state-of-the-art method iCaRL is outperformed by Generative Replay in our newly proposed scenario.

3 Foundations

While Machine Learning is about learning representations and underlying structures of large high-dimensional data sets [26], Continual Learning aims to do the same in a dynamic, non-stationary environment where the data set grows and changes over time [4]. Before studying possible solutions in this dynamic setup, we want to first mathematically establish the more general problem formulation and give an introduction to the group of machine learning algorithms that we will be dealing with: Artificial Neural Networks. We will then establish how continual learning for these neural networks can be tackled by drawing inspiration from how biological learning in humans works and by transferring it to computational learning systems [36]. We will introduce methods that have been proposed to prevent catastrophic forgetting and after introducing the setup in which these methods are commonly evaluated, we will describe in detail the generative replay algorithm that we will analyse later on.

3.1 Artificial Neural Networks

3.1.1 The Supervised Learning Problem

The largest group of machine learning algorithms are those using supervised learning. It is the class of problems where for each element in a data set we have a corresponding label providing the correct answer to the question that is to be solved. For example, if we want to train a model to recognize whether an image is showing a dog or a cat, the set of images we are training on comes with a label for each image, indicating whether one can see a cat or dog in it. Mathematically we formulate it as follows:

Given is a set of N points $(\mathbf{X}, \mathbf{y}) = \{(x_i, y_i)\}_{i=1,...,N}$, called the *training set*. $x_i \in \mathbb{R}^m$ are the input vectors or features (of dimensionality m) and $y_i \in \mathbb{R}$ are the corresponding labels. The goal is to approximate the function $f : \mathbb{R}^m \to \mathbb{R}$ underlying the data, i.e. $f(x_i) = y_i, \forall i \in (1, ..., N)$, with a model $\hat{\mathbf{y}}(\cdot, \theta)$, using a set of learned parameters θ such that:

$$\hat{y}_i = \hat{\mathbf{y}}(x_i, \theta) \approx f(x_i) = y_i$$

The parameters θ are trainable and we want to find a configuration that minimizes the distance of the model to the function. In order to measure that distance we use an appropriate loss function C, which could take the form $C(\mathbf{X}, \mathbf{y}, \theta) = ||\mathbf{y} - \hat{\mathbf{y}}(\mathbf{X}, \theta)||$ for some norm $|| \cdot ||$. Given C we can then formulate a minimization problem, which we aim to solve computationally:

$$\hat{\theta} = \arg\min_{\theta} C(\mathbf{X}, \mathbf{y}, \theta)$$

3.1.2 Multilayer Perceptrons

One method to tackle the above problem is by using perceptrons or artificial neurons as models to approximate the function f. A perceptron is a composition of a linear function, represented in the form of a weight vector $\mathbf{w} \in \mathbb{R}^m$ and bias $b \in \mathbb{R}$, and a fixed non-linear function σ, which together maps an input vector x to an output value $\hat{y}$, i.e. $\sigma(\mathbf{w}^T x + b) = \hat{y}$. The non-linearity σ is also called *activation function* and for example in the most basic case it is the unit step function, which maps any positive value to 1 and any negative value to 0 [18]. If we stack up n perceptrons of different weights and biases, we can write those parameters in a single weight matrix $\mathbf{W} \in \mathbb{R}^{nxm}$ and bias vector $\mathbf{b} \in \mathbb{R}^n$ to compute an output vector $\hat{\mathbf{y}} = \sigma(\mathbf{W}x + \mathbf{b})$ of size n, where $\sigma(\cdot)$ is applied element-wise to its input vector. If we now use the output vector $\mathbf{y}$ as an input vector for another stack of perceptrons, we create a recursion and obtain a *Multilayer Perceptron*. Since the architecture of the perceptron was inspired by how neuron cells in animal brains operate, an alternative name for this architecture is *Artificial Neural Network*. For a neural network with L layers, each layer can then recursively be defined as:

$$\mathbf{x^l} = \sigma(\mathbf{W}^{l-1}x^{l-1} + \mathbf{b}^{l-1}) \text{ for } l = 1, ..., L$$

With x^0 being the original input vector and $x^L = \hat{\mathbf{y}}$ being the final output or predictions of the network. Each vector in between describes a different representation of the original input data, which is why this method belongs to the class of representation learning methods [26]. $L-1$ is the *depth* of the network and when using multiple layers, we are performing *deep learning*, meaning that we compute multiple levels of representations [26]. In line with the supervised learning problem we can use the final outputs of the last layer to compute a loss term and adjust the model's parameters in order to minimize that loss. In the case of neural networks, the trainable parameters comprise the weight matrix and the bias vector, i.e. $\theta = (\mathbf{W}, \mathbf{b})$.

The theoretical foundation that legitimates the use of multilayer perceptrons to approximate f comes from the *Universal Representation Theorem*. It states, sloppily formulated, that if (and only if) σ is a non-polynomial function, then a multilayer perceptron of just a single hidden layer, given it contains sufficiently many artificial neurons, can approximate any continuous function f with arbitrary precision [9, 18]. With that in mind, the caveat about this theorem is that it does neither promise that there is any efficient algorithm to obtain the weights and biases of such a network, nor does it impose any limit on the number of neurons that might be necessary to achieve the desired approximation. However, using modern algorithms, namely the backpropagation algorithm, combined with newly developed dedicated computing hardware, we are nowadays in the position to train even very large-scale neural networks. Especially networks of increasing depths, i.e. networks with many hidden layers, rather than fewer hidden layers but with more neurons, have shown to be a significant improvement [25].

3.1.3 Generative Neural Networks

Common and well studied tasks for neural networks are classification and regression. In these cases, a neural network is called a *discriminative* model. Recently, *generative* deep neural networks have been proposed that differ in their learning objective and architectures from their discriminative counterparts [28]. Instead of learning the distribution of a training data set to produce output labels for new inputs, the goal of generative models is to produce outputs that take the form of data points in the original data set. But for a random input into the generative model, it is expected to generate an output that is entirely unique and different from any sample in the original data set. An example would be a model that generates portrait images of humans, which can be clearly recognized as such, but in general there exists no human that looks exactly like the person depicted in the generated portraits. The most popular strategies, of which different variants can be found in the literature, are *Generative Adversial Networks* (GAN) [13] and *Variational Autoencoders* (VAE) [22].

GANs consist of a generative model and an adversary discriminative model, which has the task to determine whether a sample was generated by the generative model or whether it actually comes from the real data distribution. The generative model essentially learns to fool its adversary and thus learns to generate increasingly realistic samples [13].
VAEs on the other hand are characterized by a single neural network that reduces the dimensionality of the input by having hidden layers with few neurons and then expands the signal again by increasing the size of the hidden layer, such that the output layer has the same dimensions as the input layer. This is similar to compression and if it were not for the non-linear activation functions this setup would be equivalent to performing a Principal Component Analysis (PCA) and reversing it again. The objective function to train this network is to minimize the difference between each input sample and its output of the network. The dimensionality-reducing part of the network is called the Encoder, the expansion part is called the Decoder. After finishing the training phase, we only use the Decoder part for generating new samples. This is done by forwarding random, low-dimensional input data into the Decoder part of the network. In order to train a discriminative network in a continual setup with generative replay, we will make use of these generative architectures. The details are described in Section 3.3.2.

3.2 Continual Learning in Neural Networks

Equipped with the foundations of artificial neural networks, we want to introduce the more dynamic setting we will be training neural networks in. Continual Learning, also called continuous lifelong learning [5, 6] or incremental learning [39, 45], refers to "the ability to continually learn over time by accommodating new knowledge while retaining previously learned experiences" [36] and has been a long-standing challenge for machine learning. In terms of our problem formulation this means that the full data set $(\mathbf{X}, \mathbf{y})$ is not available in its entirety ahead of training but instead arrives in a streaming fashion where the data distribution might change over time. Computational systems that learn new information over time tend to show a disruption or even erasion of previously learned information, that is it exhibits *catastrophic* forgetting. It has been studied that a system must be plastic in order to integrate new information but at the same time it also must be stable in order not to catastrophically interfere with previous knowledge. Since these two characteristics are directly opposed to each other, this is known as the *stability–plasticity dilemma* [15]. Even though retraining the network on the entire data set that includes all previous data together with newly available data is a solution to catastrophic forgetting, it hinders the learning of novel data in real time due to its inefficiency. An example use case that is often mentioned in the literature, also due to a recent raise in interest, is that of autonomous agents and robots which learn by directly interacting with their environment [3]. It is crucial for such systems to be able to learn and infer in real time. Humans are confronted with the same challenge and undoubtedly excel at learning throughout their life and in fact catastrophic forgetting is usually not observed in biological learning systems [12]. Therefore, as is often the case in AI research, we have turned to the ingenious makeup of our own biological compute engine to draw inspiration for possible solutions for the problem at hand. We therefore want to shortly summarize what we know about how the human brain continuously learns throughout life without forgetting the important knowledge it has acquired already (Section 3.2.1). We will then, in Section 3.2.2, transfer those insights to the machine learning setting and present different classes of effective solutions that have been proposed. Finally, in Section 3.2.3, we will introduce the established setup and benchmarks which are used to evaluate and compare the various continual learning algorithms.

3.2.1 Biological Perspective on Continual Lifelong Learning

The stability–plasticity dilemma is well studied in the human brain, which has developed ways to overcome this dilemma. The most well known learning theory involving the plasticity of neurons is *Hebbian Learning* [17]. It states simply put that repeated and persistent activation of one neuron to another leads to a strengthened connection between them. In order to model stability and prevent unbounded strengthening of few neural connections, in Hebbian systems additional constraints are imposed on such connections, e.g. by specifying an upper limit on the average neural activity. Such constraints or negative feedback to increased activity are called homeostatic [34]. Together, these two mechanisms describe how the brain facilitates lifelong learning on the neuron and synapsis layer.

When looking at the level of brain regions, there is another, additional theory about how the brain facilitates learning. On the one hand, humans have an episodic memory and can recall specific events in details and on the other hand, we can generalize experiences to form a more abstract knowledge. These two different tasks are brought together in the complementary learning system (CLS) theory [33], which locates these tasks in the hippocampal and the neocortical brain regions respectively. It states that the hippocampus learns novel information rapidly, while the neocortex is learning at a slower rate and offers for long-term retention. The interplay between these two systems allows for remembering specifics but also for learning statistical regularities. Both systems are known to deploy Hebbian Learning [38].

3.2.2 Machine Learning Perspective on Continual Lifelong Learning

We have learned that when connectionist models are exposed to new instances that deviate sharply from the previous data, catastrophic forgetting occurs. The methods that have been proposed to overcome this can be roughly categorized into three groups. They all show resemblance to the methods developed by biological systems. The first group are **architectural-based methods**. For each task they train selected parts of the networks and expand the architecture when necessary in order to represent new tasks. This method is very intuitive since with neurogenesis, there is a direct equivalent in the mammalian brain. The mammalian brain continuously grows new neurons, especially during the first developmental stages [11]. One example for this method are Progressive Neural Networks, where for any new task a new neural network is created and only the new parameters together with the lateral connections to the previous network are learned. The parameters of the previous networks remain fixed in order to avoid catastrophic forgetting [43]. A drawback for this method and

other methods that rely on a dynamic architecture is that the complexity of the neural networks keeps growing with each task [36].
The second group of strategies are **regularization-based methods**. These methods usually add an additional regularization or penalization term to the loss function of the neural network training. This essentially corresponds to implementing homeostatic behaviour, as described in 3.2.1, meaning the term acts as a counterweight to the networks unbounded plasticity. By regulating the plasticity of the network we prevent catastrophic forgetting. One example of such a regularization term was proposed with the so-called Learning without Forgetting (LwF) approach [29], where we compute and add the distillation loss to the total loss. The distillation loss is a measure for how similar the old network is to the newly updated network and by trying to minimize this loss, the network is incentivized to only minimally change its parameters when learning the new task. Existing regularization methods have shown to perform well on simple scenarios where the task to be solved is known at interference, but they perform poorly when this is not the case.
The third group, which at the same time is the group our research focuses on, are **rehearsal-based methods**. Essentially, these methods use some form of memory replay that is used to remind the network of the knowledge it has previously learned and and therefore to reinforce it [36]. This approach shows parallels with the complementary learning systems we have mentioned before, where we have the interplay of two components, one of which represents a more detailed episodic-like memory and the other component generalizes and learns statistical patterns. Rehearsal-based approaches have been proposed early on in the research of neural networks [40] and a simple example would be Exact Replay. In this case, we store a manageable subset of the previous data and interleave it together with new data when we continue the training of a network. We will go into more detail for another rehearsal-based method, Generative Replay, in Section 3.3.

3.2.3 Continual Learning Scenarios

Our terminology will mainly adhere to the framework proposed in [50] for evaluation of continual learning algorithms. The authors propose three continual learning scenarios that can be applied for various benchmark data set. This framework has been widely accepted and is being used throughout the literature. We want to introduce the scenarios and benchmarks here, together with definitions of the basic terminology that we are using in this work.

In continual learning a model learns different tasks and these tasks are assumed to be clearly separated and learned sequentially one by one. A *task* is defined by a data set and corresponding labels, where either the set of labels or the data itself has a unique distinction to separate it from other tasks. A task could for example be to learn another Atari game after a model already has learned a first. Or it could mean that a model that has been trained to distinguish images of cats and dogs now learns to also distinguish images of elephants as well. The learning phase of such a new task is called *experience*. We will argue in Chapter 6 that the assumption of clearly separated tasks where in each experience the model sees the full data set belonging to a single task only is too restrictive. For now, and in particular in Chapter 5, we will

stick to this understanding of the CL setup. A *scenario* describes a set of experiences, and is defined by the concrete tasks that are learned in each experience.
The first type of scenario is where during training and inference the model receives information about which task is to be solved, meaning that each data point comes with a *task label*. We will refer to this type of scenario as *task-incremental learning* or task-IL. It is a very common scenario as many reinforcement learning problems are structured this way [49], but it is not realistic for other applications. If no task labels are given, we distinguish between two scenarios. The first case is *domain-incremental learning* or domain-IL, where no task labels is provided and the model does not need to infer the task label during inference. This would usually be the case because the different tasks are all structured in the same way, such that the output takes the same form and only the input changes. An example for this would be after classifying restaurant reviews to be either positive or negative. The next task, or domain, could then be to also classify film reviews into positive or negative. The possible outputs of the network stays the same but the input distribution changes. Domain-IL is somewhat more difficult than task-IL, however its possible applications are still limited such that is has gotten less attention compared to the third class of scenarios: *class-incremental learning* or class-IL. Here, we do not provide task labels with the data points either. However, the possible outputs of the network are different to each other. The model is required to infer which task it is seeing in order to succeed. This is exactly the case where a model has learned classification for a set of classes and then needs to distinguish a new class, like images of elephants as mentioned in the above example. Architecture-based approaches as well as regularization-based approaches have been shown to work well in the task-IL scenarios but do not extend well to domain-IL and class-IL. Class-IL generally is seen as the most challenging CL setup [50].

Common benchmarks can usually be categorized to belong to one of the three groups of scenarios. In this work, we will focus on class-IL. One of the most commonly used benchmarks for this type of scenario is *splitMNIST*. The underlying data of this benchmark is the *MNIST* [27] data set which consists of 70,000 grayscale images with 28x28 pixel each. Every image shows a single handwritten digit ranging from zero to nine. The images are separated into 60,000 training images and 10,000 images for testing and they are roughly uniformly distributed among the ten classes of digits. This data set is very widely used for testing and benchmarking machine learning algorithms, for which case the images are usually shuffled prior to training. Conversely, for the splitMNIST scenario, we split the data into ten experiences, each of which contains all the images of a single digit. We then let the model train the experiences sequentially with each experience containing a single task.

3.3 Fundamentals of Generative Replay

Rehearsal-based methods (also called *replay*-based methods) have shown to perform very well in the continual learning setup described above [50]. In fact, the current state-of-the-art approach, namely iCaRL [39], is rehearsal-based. However, there are some drawbacks attached to relying on storing original data points for rehearsal: there are cases where it is not possible to store these points until we want to update our model. This could for example be for storage reasons when the original data set continuously grows or for data privacy reasons, where data is only allowed to be kept for a limited amount of time. And indeed, shortly after the first rehearsal methods have been proposed in the early nineties [40], the same researcher worked on finding a solution for their aforementioned drawbacks. In [41], Robins introduced the concept of pseudo-rehearsal where essentially randomly filled input vectors, together with the outputs that are produced by forwarding them through the neural network, construct a new *pseudo* data set, which can be used as replay data. Despite the randomness of the replay samples, this approach proved to reduce the severeness of the catastrophic forgetting. But it didn't solve the problem entirely. Despite this early success, it took more than two decades until Hanul Shin combined the major advances that have been made in data generation techniques with the pseudo-rehearsal approach. These advances are mainly associated with the rise of deep learning, where dedicated neural networks are trained whose output takes the shape of the data points of the input data set. Recently for instance, these networks have been shown to generate realistic but previously unseen images [13, 22]. In [44], Shin proposes to train such a generative model alongside the classifying model and regularly generate data resembling the original data set and to interleave it with the new data that is to be learned. The method was named *Generative Replay* and it subsequently sparked related works to build up on the idea [49]. In this chapter we will first define the scenario we will be operating in (3.3.1) and then rigorously explain the GR algorithm (3.3.2).

3.3.1 Scenario for Generative Replay

The problem we are trying to solve is a classification task in a class-incremental learning scenario (as defined in Section 3.2.3). That means we have a series of K experiences each defined by a data set $(\mathbf{X}, \mathbf{y})^{(k)} = \{(x_i^{(k)}, y_i^{(k)})\}_{i=1,...,N}$. Where $x_i \in \chi$ is the ith data point in experience k, e.g. a gray-scale image, and $y_i \in \mathcal{C}$ is the corresponding class label from a set $\mathcal{C}$ of classes. If we let $\mathcal{C}^{(k)}$ be the set of labels encountered in experience k, then what defines the class-incremental scenario is the fact that in general $\mathcal{C}^{(k)} \setminus (\mathcal{C}^{(k-1)} \cup ... \cup \mathcal{C}^{(0)}) \neq \emptyset$, meaning that in experience k there are classes that have not been seen by the model before. The property that leads to catastrophic forgetting on the other hand, and which makes the task difficult, is that there are classes in an experience k that will not be encountered again in any experience $i, i > k$. In common benchmark scenarios, like the splitMNIST scenario [51], this is the case for all classes in an experience, amounting to $\mathcal{C}^{(k)} \cap \mathcal{C}^{(k')} = \emptyset, \forall k \neq k'$. We will later argue that this assumption is not realistic and there are indeed algorithms, most prominently iCaRL, that rely on this assumption and perform much worse if it is not given (see Section 6).

3.3.2 The Generative Replay Algorithm

We now define the *scholar* as a tuple $S_k = (\text{Model}_k, \text{Generator}_k)$ consisting of a classifier model, which we will either call *solver* or simply the *model*, and a generative model, which we will call the *generator*. We will train a sequence of scholars where S_k is trained on $(\mathbf{X}, \mathbf{y})^{(k)}$ and the input data is augmented by interleaving it with generated data $(\hat{\mathbf{X}}, \hat{\mathbf{y}})^{(k)}$, produced by Generator_{k-1}. The entire training paradigm we employ in our experiments is described in Algorithm 1 and visualized in Figure 1.

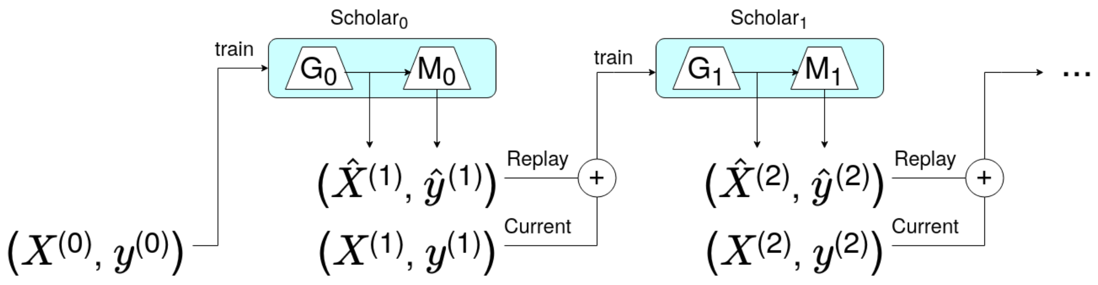

Figure 1: Training a sequence of scholars. In the first iteration we train the model M_0 and generator G_0 on the data set of the current experience. In subsequent iterations the current input data is interleaved with replay data and used to update the scholar. Questions regarding how much replay data is generated and how it is best combined with the real data will be asked and answered in chapter 5. Another consideration that can be made is whether we generate all necessary replay data before the training phase or whether we generate the data in every training iteration. The latter version benefits from not having to store a potentially large replay data set, however it requires to actually store a copy of the old scholar model (instead of overwriting it directly).

In practice we do not store the intermediate scholar models but simply overwrite S_k with S_{k+1} when training the next experience. This is because our goal is to obtain a single scholar S_K that contains all the knowledge of $S_0, ..., S_{K-1}$ and additionally performs well on $\mathbf{X}^{(K)} \cup ... \cup \mathbf{X}^{(0)}$. Furthermore, outside of research, we would not have access to the sequence of experiences beforehand, but the experiences would become available with a temporal distance. In Algorithm 2, we have summarized the steps to update a previously trained scholar S when new data $\mathbf{X}$ becomes available.

4 Implementation Process

The implementation of the generative replay algorithm took a central role in creating this master thesis, both time-wise and conceptually. There are several reasons for that. For one, the implementation is crucial to all following experiments that build upon the basic GR algorithm. It was important to create a piece of software that is not only effective and efficient, but also easily extensible and adaptable. Many questions only came up during the implementation phase itself and it proved vital to have prepared for previously not anticipated use cases, by writing modular and extensible code. In fact, while implementing, we actually noticed that depending on the details of the implementation, the performance of the algorithm can change under otherwise equal conditions. Chapter 5 is a result of this process and our program can perform any of the proposed variations by simply specifying an input value. Upon researching for existing code bases, we noticed that existing public repositories were rigid and not easily adaptable to our needs. On the other hand, we found that the open-source continual learning framework Avalanche was designed for rapid prototyping, evaluation of many methods on easily changeable scenarios. However, while a range of CL algorithms had been implemented for the framework, we found Generative Replay to be missing among them. We made it to our goal to add a user-

Algorithm 1 Generative Replay Algorithm

```
Input: $X^{(0)}, ..., X^{(K)}$: data for each experience; $y^{(0)}, ..., y^{(K)}$: corresponding class labels
  Model$_0$ $\leftarrow$ random initialization
  Generator$_0$ $\leftarrow$ random initialization
  Model$_0$ $\leftarrow$ train($Model_0, X^{(0)}, y^{(0)}$)
  Generator$_0$ $\leftarrow$ train($Generator_0, X^{(0)}$)
  for k $\in$ [1, ... , K] do                                ▷ K is the number of experiences
      $\hat{\mathbf{X}}$ $\leftarrow$ generateFrom(Generator$_{k-1}$)
      $\hat{\mathbf{y}}$ $\leftarrow$ Model$_{k-1}$($\hat{\mathbf{X}}$)
      $\mathbf{X} \leftarrow \mathbf{X}^{(k)} \cup \hat{\mathbf{X}}$
      $\mathbf{y} \leftarrow \mathbf{y}^{(k)} \cup \hat{\mathbf{y}}$
      Model$_k$ $\leftarrow$ train(Model$_{k-1}$, $\mathbf{X}, \mathbf{y}$)
      Generator$_k$ $\leftarrow$ train(Generator$_{k-1}$, $\mathbf{X}$)
  end for
```

Algorithm 2 Updating a Scholar with Generative Replay

```
Input: $\mathbf{X}$: newly available data; $\mathbf{y}$: corresponding class labels
Input: Trained scholar $S =$ ( Model, Generator)
Output: Scholar $S =$ ( Model, Generator) with knowledge of input scholar and $(X, y)$
  $\hat{\mathbf{X}}$ $\leftarrow$ generateFrom(Generator)
  $\hat{\mathbf{y}}$ $\leftarrow$ Model($\hat{\mathbf{X}}$)
  $\mathbf{X} \leftarrow \mathbf{X} \cup \hat{\mathbf{X}}$
  $\mathbf{y} \leftarrow \mathbf{y} \cup \hat{\mathbf{y}}$
  Model$\leftarrow$ train(Model, $\mathbf{X}, \mathbf{y}$)
  Generator$\leftarrow$ train(Generator, $\mathbf{X}$)
  return ( Model, Generator)
```

friendly flexible implementation of the GR algorithm to the Avalanche framework and to make the framework the center piece of our experiments as it allows for the creation of custom scenarios and comparison with other methods. Next, we will give a short overview of the Avalanche framework, afterwards we share the difficulties we faced and the highlights of the implementation and finally conclude by describing the artifacts that came out of this process and where they can be found.

4.1 Avalanche Library

Avalanche is an open-source end-to-end library for continual learning research based on PyTorch, that was proposed in response to growing interest in continual learning [31]. It aims at eliminating the difficulties of re-implementing and porting existing algorithms to new settings for evaluation and comparison. Avalanche uses an architecture consisting of templates for strategies and plugins to add additional functionalities. A template defines an interface for training and evaluation of a provided model on a data set, such a template could for example implement a supervised learning training cycle. Plugins can simply be added to templates and each plugin implements one or more *callbacks*. A callback is essentially a function that is called at a specified step in the training cycle. For example, if we want to update our generative model after training the discriminative model on an experience, we can do so by specifying it in the "after_training_experience" callback, which will be executed after the training of each experience.

4.2 Difficulties

While the Avalanche library promised to assist us with our research, it nevertheless came with a steep learning curve as the library was still at an early stage such that documentation was incomplete and not always up-to-date. To make our strategy reusable and implement it in accordance to the library's architecture, we had to get familiar with the library first. The library is at such an early stage, however, that during our implementation a restructuring took place as well as the first *beta* release. Working on such a young project comes with its own set of problems, as many things are still subject to change. The strategy and plugin architecture provide a framework for developers that is very versatile and flexible, albeit it is not always straightforward to see how to implement specific requirements within this framework. In fact, there have been other attempts before, most notably by the author of [49, 50], to implement generative CL methods in the Avalanche Framework, which have been struggling with the Avalanche strategy architecture [19]. As the framework matures and documentation gets more solid, this should become less of a problem.

4.3 Highlights

We eventually raised a pull request that underwent a several weeks long period of scrutiny, change requests and adaptations from our site. The process required endurance but was rewarding as the contribution was eventually accepted and our implementation is now part of the python library Avalanche and has been published in the Beta 0.2.0 release.

After this positive outcome, we eventually were able to yield the results of our work. We found that running the algorithm on different scenarios and with various variations, as we do in Chapter 5, works efficiently and robustly. Furthermore, we have created a starting point for anyone who wants to conduct further research on CL and Generative Replay. It would be straightforward to add extensions, such as the ones proposed in [49], or to run and compare Generative Replay with other methods on new scenarios and data sets. We believe that our contribution substantially lowers the bar of entry required for anyone to further research on Generative Replay.

4.4 Artifacts

The implementation of the GR algorithm is open-source and can be accessed via the Github repository of Avalanche. A history of the commits alongside the discussion and change requests of the Avalanche team can be viewed on the website of the pull request [7]. The usage of the strategy follows those of other methods implemented in Avalanche and there are examples of how to continually train a scholar model or a standalone generative model provided in the libraries examples section. A documentation of the implementation itself is provided by the extensively commented source code.

Using this implementation as a basis, we designed and executed several experiments for Capters 5 and 6. For each experiment we created a separate Jupyter Notebook and added all of them to a public Github repository for reproducibility [21].

5 Experiments: Understanding Generative Replay

From hereon we will describe in detail the experiments we have run and we will explain the design decisions we took that guided our research and which helped us to circumvent the challenges we faced during the implementation and execution.
Task: While CL is gaining importance in many AI fields, such as reinforcement learning or language models, we focused on the case of **image classification** and, as a necessity for the employed algorithm, **image generation**.
Dataset: All experiments are be based on the hand-written digits data set **MNIST**, as it is sufficient to allow us to evaluate the effectiveness of various CL methods on it while also being small enough to allow the conceived experiments to run within our limited compute power and within the time frame of this thesis. Furthermore, many other works use it as benchmarking data set as well as for evaluating newly proposed strategies [44, 50, 51]. Good results on the MNIST data set are, however, not to be mistaken to hold generally true on more complex data sets and data sets of different input types. It has been observed that the CL problem becomes much harder when it is subject to a more complex data set or when substantially increasing the number of possible classes to be learned in the class-IL setting [49].
Goal: Our research is to be understood as a foundation to better understand the GR algorithm and a first try to establish a more realistic benchmark to evaluate CL methods. In order to verify the robustness and readiness to use in production of such methods, further research needs to be done on scaled scenarios deployed on respec-

tively scaled computing resources.
Hardware: In our case, we use an **Intel Core i7-5600U CPU @ 2.60GHz** for which the training of a single scenario in most cases took between **15-20 minutes**.
Architectures: The model architectures for the scholar (i.e. generator and solver) are shown in Figures 2 and 3. They were deliberately kept simple to reduce training time and they were both implemented as PyTorch models.
Metrics: Our main metric for evaluation will be the accuracy of the solver model on the 10k image MNIST test data set, that was previously unseen during training. We measure the "forgetting" of a class by the change in accuracy over time (i.e. between experiences) and consider a class to be catastrophically forgotten when the class accuracy drops close to zero ($< 5\%$).

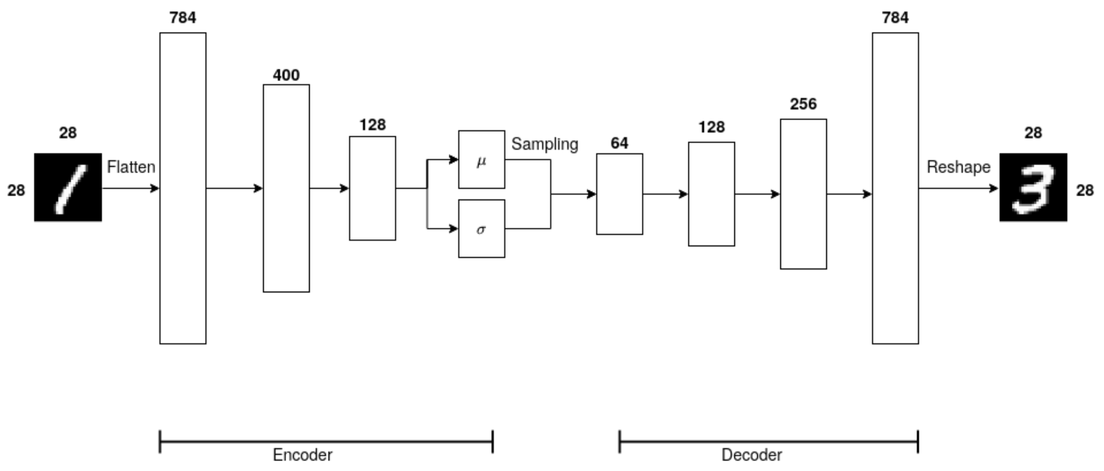

Figure 2: Schema of the generator's architecture we employ in all our experiments. We use a Variational Auto Encoder, which can be separated into an Encoder and Decoder part.

5.1 How We Will Try to Understand GR

Our **first goal** was to understand the intricacies and inner workings of the Generative Replay algorithm. We will base our research on [44], where the base algorithm was proposed for the first time. However, in this paper, a row of questions were left unaddressed when it comes to the exact implementation as well as a discussion regarding under which circumstances GR works best and how to appropriately set the adjustable parameters. For example, a first obvious question when discussing *Generative* Replay is that of the generator's architecture. The authors of [44] made a not further justified choice of a GAN as their generative model, whereas [49] chose a VAE at least in parts because some of their proposed improvements of the GR algorithm make use of the VAE architecture. In our own initial exploration phase, we found that the GAN training at times suffered from mode collapse, required a substantially greater amount of training time and the training in general was harder to control. Because of this experience and because we kept the possible improvements of [49] in mind, we decided to opt for a VAE. Our implementation is structured in a modular way such that the generator can easily be swapped. However, all experiments pre-

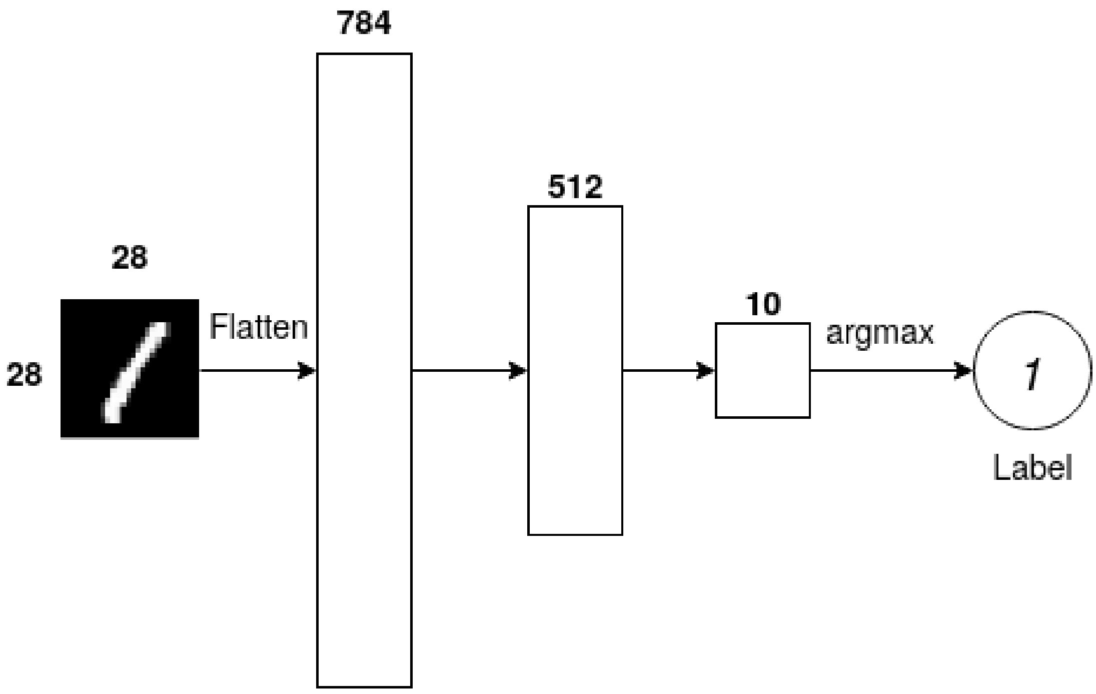

Figure 3: Schema of the discriminator's (or solver's) architecture we employ in all our experiments. We use a simple fully connected model of a single hidden dimension of depth 512.

sented in the following sections use a VAE as generator.
Experiment 1: Another open question that has come up during the implementation phase and that has proven to be central to the performance of the algorithm is how much replay data to generate and exactly how it is best to be combined with the current training data. We want to discuss this question in greater detail and observe how it affects the performance of the solver. In the following subsection 5.2, we will compare five different variations, in terms of the amount of replay data used and how the loss on the replay data is weighted relative to the real data. We will see that it is an important question to consider before implementing GR.
Experiment 2: Furthermore, in the subsection after that, Section 5.3, we want to address one of the known shortcomings of the GR method which is that much of the complexity of the original problem is shifted to the generator since generating clear, realistic images is considered a difficult task on its own. We will see that even though GR does not require perfect replay images (perfect for the human eye) the generator does in fact play a central role for the method and acts as a kind of bottleneck to the upper limit of the solver's performance.

5.2 Various Forms of Replaying Data

If we look at Figure 1 we see that we use the generator to generate replay data X' and compute the losses for both X' and the real samples X. We then combine the losses before performing the optimization step. This provides us with two adjustable parameters, one is the amount of replay data that is generated, i.e. the cardinality of X', and the second is the weighting factor λ that we apply when summing up the losses $\mathcal{L}_{data}$ and $\mathcal{L}_{replay}$ to compute the total loss, as seen in Equation 1. Both presumably let us adjust the importance of the replay data relative to the real data. Say, if we weigh the loss of the replay data twice as much as the loss for the real data (i.e. $\lambda = \frac{2}{3}$), we would expect to find a bias in the solver's accuracy towards the classes of the replay data. Similarly, increasing the number of generated images would yield the same effect.

$$\mathcal{L}_{total} = (1-\lambda)\mathcal{L}_{data} + \lambda\mathcal{L}_{replay} \tag{1}$$

In the following, we look at various performance metrics for different settings of these two parameters. We will evaluate the approaches on the class-incremental *splitMNIST* benchmark scenario, which we described in Section 3.2.3. For simplicity, we have ordered the experiences numerically ascending, meaning that the model first encounters all images of zeros and lastly all the images of nines. Since our main goal is to overcome catastrophic forgetting, which is defined by the drop of the accuracy for classes seen in previous experiences, we will use the accuracy per class as our main metric of success. We will look at further metrics that can guide us in how to tweak the algorithm to gain higher accuracies and reduce forgetting.

5.2.1 Vanilla/Default Approach

In a first and naive approach, without any further knowledge about how much data to generate or how to weigh the loss terms of replay and real samples, we simply use as many replay data points as there are real data points in each batch, i.e. $|X| = |X'|$. Furthermore, we resorted to simply adding the two loss terms without any additional factor, i.e. $\mathcal{L}_{total} = \mathcal{L}_{data} + \mathcal{L}_{replay}$. We define this as the *default* setting of the algorithm. Using this form of generative replay in the splitMNIST scenario gives us an overall accuracy for the solver model of 59.25%, after training the scholar model on all ten experiences with five epochs per experience and a batch size of 64. This number is relatively low when compared to the upper bound of 95.32% which is obtained by training the solver on the shuffled MNIST data, effectively avoiding catastrophic forgetting all together. When we look at the respective accuracies per class in Figure 4, we see that catastrophic forgetting takes place and after the last experience, the class three has been forgotten entirely (0%), dragging down the overall accuracy (the overall accuracy describes the average over all classes). Moreover, when looking at the replay samples that were generated by the generator per class in each experience, with a subset plotted in Figure 7, we see that not only were there no replay samples for classes one, two and three mixed into the training data of the last experience, but furthermore do the replay samples of other earlier classes, e.g. class zero or four, not necessarily correspond to their respective class, at least not to the human eye. This suggests that not only the solver but also the generator is still suffering from catastrophic forgetting. In order to quantify this catastrophic forgetting for the generator, we can look at the distribution of classes among the replay samples. A generator trained on the entire balanced MNIST data set generates samples which are on average again following the same balanced distribution of the original data set. For successfully training a generator in the splitMNIST scenario, we therefore want to achieve the same: ideally, by the end of each epoch, we want the generator to produce on average the same amount of images of all previously seen classes. When looking at the actual distribution of the generated samples in Figure 6, we see that until experience four roughly the same amount of replay images for each class is used. However, in the experiences after that, we see a strong bias towards the classes seen in the most recent experiences. A more condensed form of this can be seen in Figure 5.

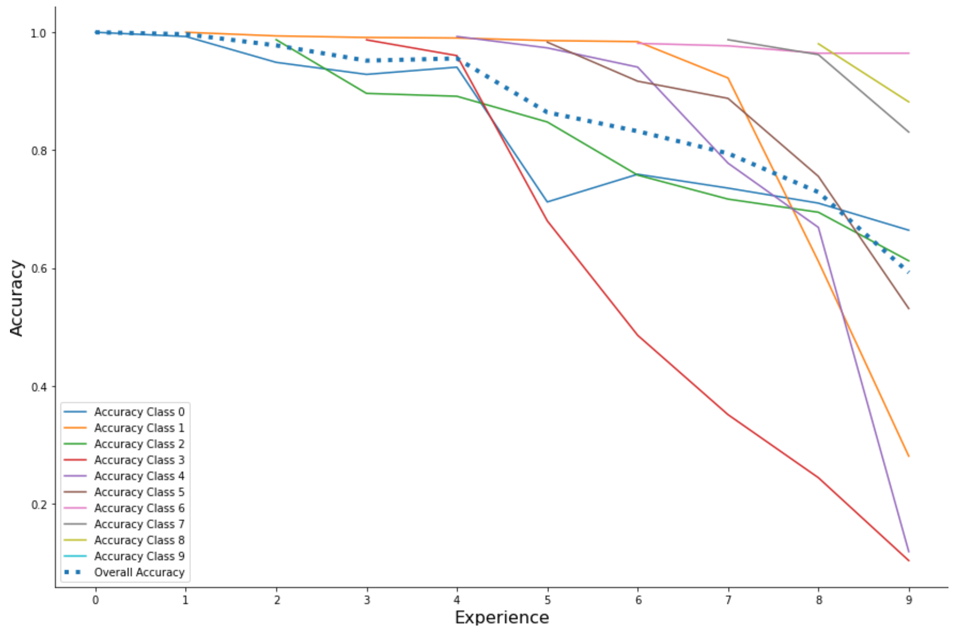

Figure 4: **Vanilla Approach:** Results of the default GR implementation. Accuracies for each experience of the splitMNIST scenario, where each experience shows the accuracy for every class that has been encountered until that point plus the average over those classes.

5.2.2 Fixed Replay Size

From the results of our vanilla implementation, we see that although we alleviated the catastrophic forgetting to some extent, we can still observe a decay in the accuracy of classes from earlier experience. We want to try and nudge the optimizer towards putting more importance on previously seen classes. A simple way of achieving this could be by generating more replay samples than there are real data samples, such that the new data is relatively less important in the computation of the total loss and therefore in the optimization step. Concretely, we implement this by setting $|X'| = 400$ while keeping $|X| = batchsize = 64$. For these parameters, we see a jump in the final average accuracy over all classes of about 10% reaching 75.49% as seen in Figure 14. And indeed, we are able to observe the desired effect that the generator, even in the last experience, remembers classes seen in the first experience. Also, the set of replay samples is more balanced than before. In fact, we can even see a bias towards the first class (Figure 16). However, the replay samples especially for some of the classes in the middle are not exactly recognizable with class three not showing up in the replay set

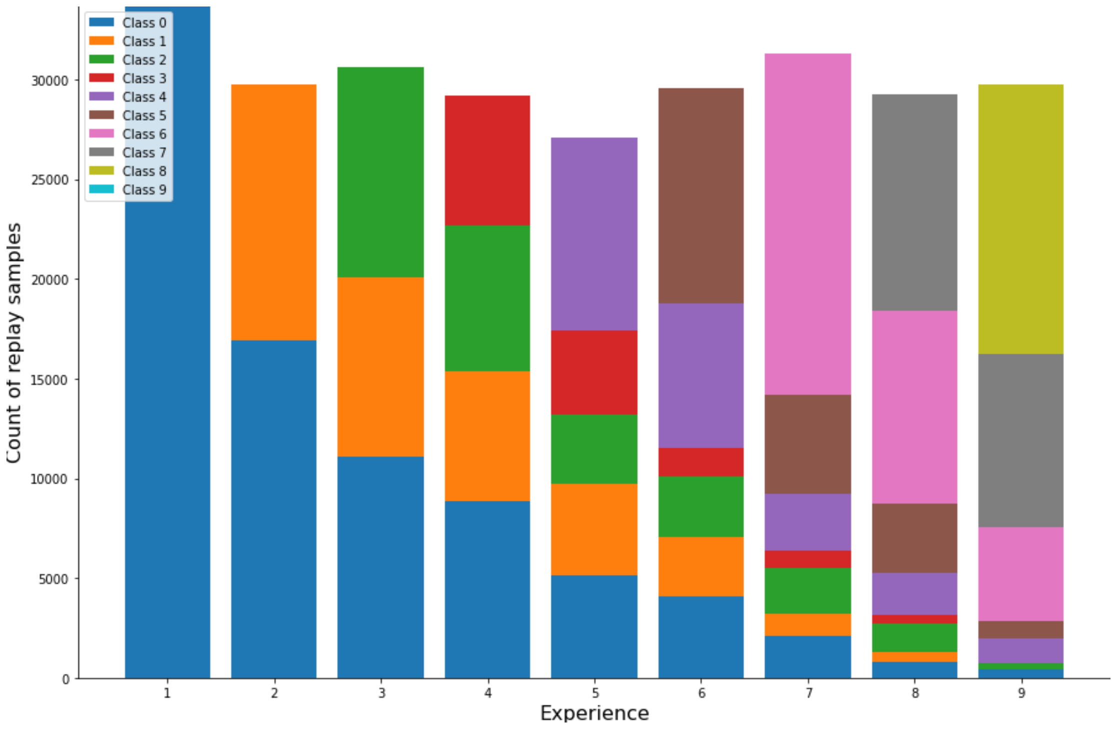

Figure 5: **Vanilla Approach:** Distribution of replay labels among the ten classes. Plotted in a single bar for each experience (starting from experience 1; there are no replay samples used in experience 0).

at all (Figure 17). Since this approach worked well for the first classes of the scenario but then dropped in performance again, we want to look at an approach where we increase the replay set size gradually as we train more experiences. We hope that this will enable the generator to remember all classes.

5.2.3 Continually Increasing Number of Replay Samples

Here, the idea is to double the amount of replay data $|X'|$ with each new experience. This means that in each training iteration we obtain the number of replay samples to generate by Equation 2 while the number of real data stays at constant: $|X| =$ batchsize.

$$|X'| = k * \text{batchsize} = k * 64, \text{ where k denotes the current experience} \tag{2}$$

Running the same splitMNIST scenario as before, we can now see in Figure 20 that at the end of the training every class still is remembered by the generator and, except for a bias to the first two class, we find that for the other classes roughly the same number of replay data is used during the training. In Figure 19, we have compiled these statistics in a single bar plot to visualise the increasing amount of the total number of replay samples per experience. And indeed, this is translating into better accuracies, obtaining an average among all classes of 83% (Figure 18). Unfortunately, it is easy to see that this approach is not scalable for the case of more complex data

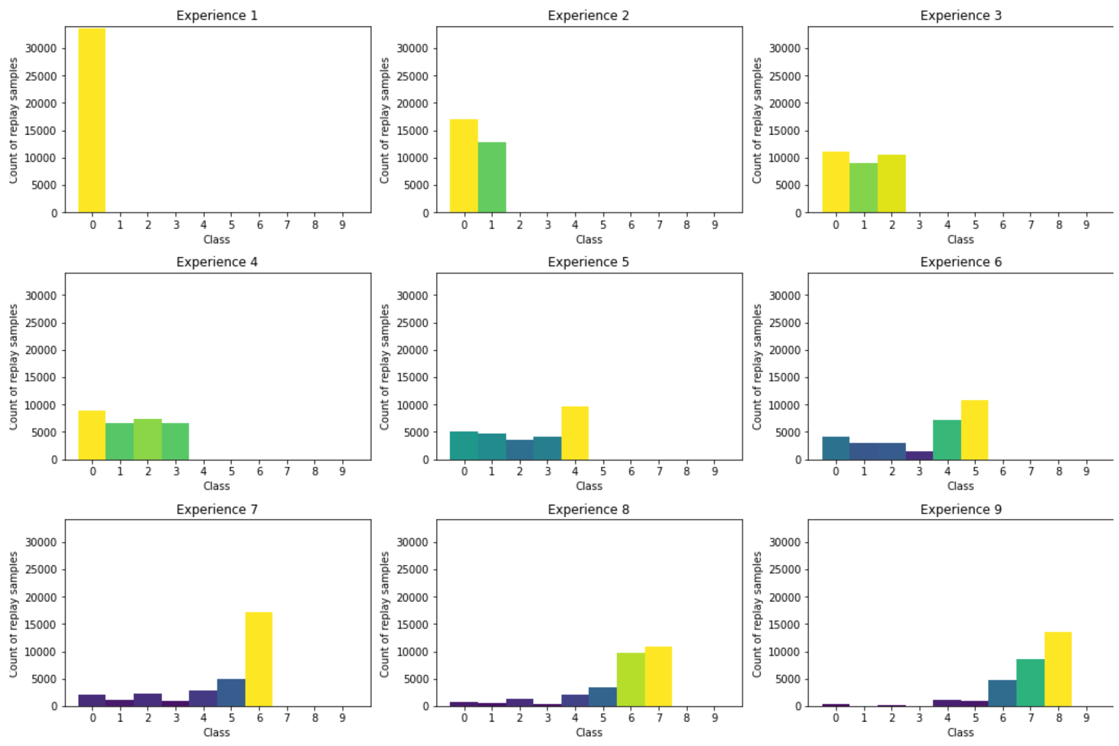

Figure 6: **Vanilla Approach:** Distribution of replay labels among the ten classes. Plotted in a separate bar plot for each experience (starting from experience 1; there are no replay samples used in experience 0).

sets containing many more classes as we would be handling a substantially increased amount of data with each new experience. Even in our simple example, in the tenth experience we are already computing the losses of roughly 250,000 replay samples - opposed to just 60,000 samples in the entire training data set (Figure 19). Instead of increasing the importance of past classes by increasing the number of replay samples, we could also more directly manipulate the loss function. By splitting up the loss function into two terms, one for the real data and one for the replay data, we assume that we can directly adjust the importance of the replay data without any additional computational overhead. This is what we will try in the next section.

5.2.4 Weighted Loss

In a similar fashion as above, in each experience, we increase the relative weight of the loss $\mathcal{L}_{replay}$ on the replay data that we add to the loss on the real data to compute the total loss term in each training iteration. Specifically, we set $\lambda = \frac{k-1}{k}$ (thus $(1-\lambda) = \frac{1}{k}$) when training the k-th experience, where λ is the weighting factor used in the loss computation in Equation 1. This approach does not incur in any additional storage or computing requirements. Nonetheless, we are able to obtain results similar to the approach of increasing the amount of replay samples from Section 5.2.3. The accuracies are plotted in Figure 22 and the distribution of replay samples among the classes can be seen in Figure 24. Even though the distribution is not perfectly uniform, we still find that the generator is able to remember all classes it has encountered

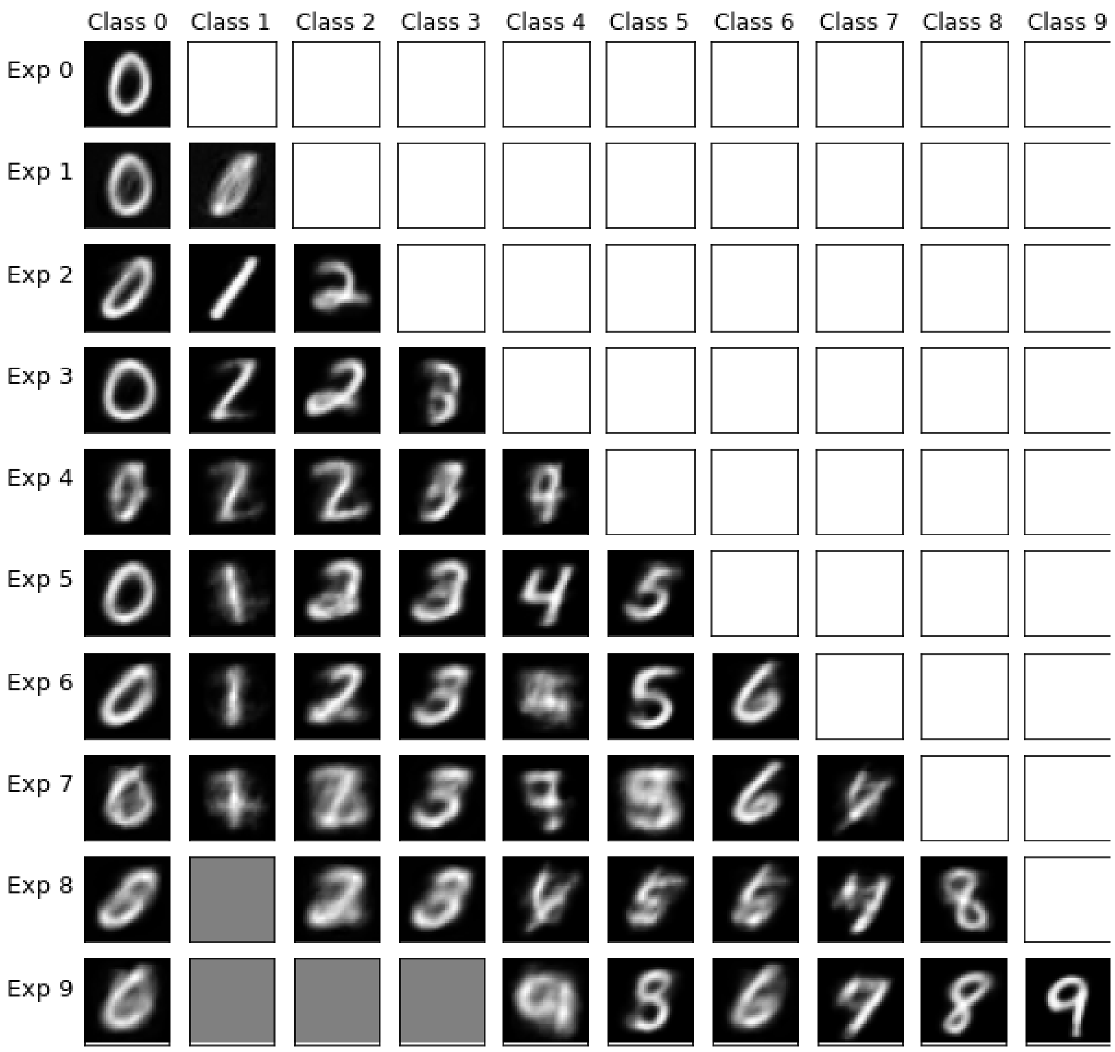

Figure 7: **Vanilla Approach:** Samples of images generated for each class after training an experience. White tiles mean that this class has not been encountered yet, grey tiles indicate that a class has been forgotten.

during its training (Figure 25). The average accuracy is similar as before with 81.38%. Interestingly, when implementing the GR algorithms, there is a decision to be made, whether to generate all replay data before each experience (and thus not having to store additional copies of the generator and model) or to generate them on-demand in each training iteration using a copy the the final generator and model from the previous experience. We initially implemented the first but then shifted to the latter in order to not having to store a huge amount of replay data. Before this change, with otherwise the same settings, we obtained an accuracy of 75.61% (i.e. a similar accuracy as in the case of a fixed replay size of 400). By not reusing the same replay data in each epoch, the solver gets to see a greater variety of samples during the span of each training phase and this fact seems to be, at least in part, responsible for the better overall accuracy. Furthermore, this would also explain, why the weighted approach is still faring a bit worse than the increasing replay size approach, since with the latter one the solver and generator get to see many more replay data in total.

5.2.5 Balanced Replay Samples

We have identified that in order for a solver to maintain a good accuracy for a class, the generator of the scholar model needs to be able to remember and reproduce images of that class. Ideally we would like the generator to generate a uniformly distributed set of samples (given that the original data set is uniformly distributed). We have found ways how to get closer to this goal, but we have no direct control over the distribution of the replay data set. Our generator is not conditional meaning that we cannot condition it to generate an image of a specific class on command. In order to estimate how much of a gain a truly uniform replay data set would provide, we could employ a conditional generator and see how this influences the accuracy. To keep it simple, we instead train separate generators for each experience such that each generator can reproduce images of a single class. This nonetheless gives us control over exactly how many replay images per class we want to add to the training batch, without having to go into the details of actually implementing a conditional generator. Furthermore, it gives us the opportunity to correctly label the replay data instead of relying on the solver to generate those labels as it is done in our base algorithm (see Figure 1). The quality of the labels in the original algorithm is bound by the quality of the solver itself and this potentially left us in a negative feedback loop: incorrectly labeled replay samples would distort the training and lead to a lower accuracy of the solver. A lower accuracy of the solver in turn would then produce labels that are less accurate. We now have the chance to also quantify, next to the effect of balancing the replay data, the effect that this feedback loop has. Indeed, with a balanced replay data set (see Figure 28), we are able to reach the highest accuracy among all proposed approaches with 82.75%. But even though all replay images are recognizable and every class is represented in the replay data (see Figure 29), this accuracy is still below the theoretical threshold. Without using our knowledge to improve the quality of the labels and letting the solver produce the labels for the replay images, we reach an average accuracy of 81.99%, which is only a difference of less than one percentage point.

5.2.6 Summary

The vanilla implementation as conceptionalized in Figure 1 did indeed prevent complete forgetting of previous classes, but it was not able to scale to the entire set of classes. We therefore looked at various approaches that helped us to increase the overall accuracy and to further reduce the forgetting. In order to get an understanding of how much better these approaches fared compared to naively training the model without any Generative Replay, we have plotted the accuracies of naively training our model in the splitMNIST scenario in Figure 8. Almost immediately, the model forgets all classes it has seen in previous experiences and simply predicts any image to belong to the class of the current experience. It is therefore evident that our GR implementation constitutes a substantial improvement. On the other hand, to see how much room for further improvement there is and to see what the simple model itself is capable of, we have let our model train on the traditional MNIST dataset, i.e. training a single experience containing all classes from zero to nine. This is often also called *joint training* and in this case our model reaches an accuracy of 95.32%. We have compiled the final accuracies of all our approaches together with the naive and joint training in Table 1. We see that increasing the number of replay samples, calculating a weighted loss term or using a balanced set of replay samples all produce similarly good results. However, we can also see that the model itself is in theory able to perform much better (95.32%, when using joint training) than our best run with GR (82.75%). We will therefore take a closer look at the generator and the limiting factors of Generative Replay in the following section (Section 5.3). Which variant of the GR algorithm to choose might finally depend on the underlying scenario. From our results here, a conditional generator that allows to generate class-balanced replay data would be the best choice. However, the list of variants we compared here is by no means exhaustive. For example, for the variant of weighting the losses, we could conceive many more sequences of λ than we did here. Nevertheless, we have achieved good improvements and our accuracies are above those that are mentioned in some survey papers. [28], for instance, reports 79.38% on splitMNIST for a model using GR and a VAE as a generator. The authors did conceal their exact implementation of the algorithm.

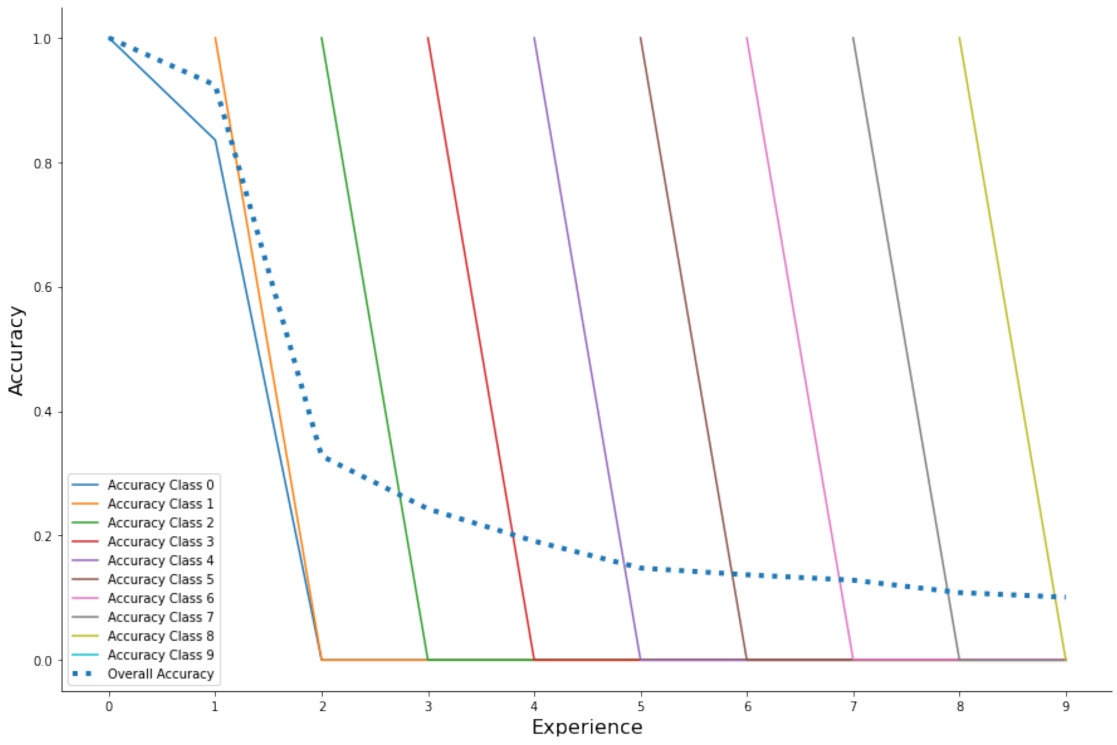

Figure 8: **Naive training without Generative Replay:** Accuracies for each experience of the splitMNIST scenario, where each experience shows the accuracy for every class that has been encountered until that point plus the average over those classes.

For simplicity, we will use the fixed replay size approach in our further experiments. This is because we have seen it offers an acceptable accuracy and a general applicability without any further conditions. The accuracy is acceptable since we are not interested in merely maximizing the performance but rather are we interested in understanding the algorithm better, for which a reasonable accuracy is sufficient. And the general applicability of the approach is opposed to the other variations, which either do not scale well to scenarios with many experiences (5.2.3 and 5.2.4) or which would require the generator to be conditional (5.2.5). The fixed replay size approach stays the same independent of the underlying scenario.

Variation	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Class 8	Class 9	**Average**
Vanilla	66.43	28.10	61.24	10.40	11.91	53.14	96.45	83.07	88.19	98.12	**59.25**
Fixed Replay Size 400	94.90	98.94	73.93	6.14	61.00	58.86	94.15	87.94	81.21	94.15	**75.49**
Increasing	92.96	98.94	76.07	67.62	65.17	63.23	91.23	89.01	83.26	93.16	**82.46**
Weighted loss	86.22	99.30	75.97	67.92	62.22	64.46	94.99	83.46	82.55	92.96	**81.38**
Balanced Replay	90.20	96.12	81.59	76.24	54.89	78.25	94.68	70.33	76.59	99.01	**81.99**
Balanced w exact labels	90.61	97.27	84.11	76.63	58.66	77.24	94.26	73.15	74.44	98.71	**82.75**
Naive training	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.00	0.00	100.00	**10.10**
Joint training	98.47	98.06	94.38	95.25	95.42	93.72	96.35	94.07	93.94	93.16	**95.32**

Table 1: Accuracies in percentage (%) per class and the average over all classes for various training paradigms after training on all data points

5.3 Generative Replay Repeated over Itself

We have seen that when using joint training, our model achieved an average accuracy of above 95%, our best generative replay approach fared some 10% below that. We want to understand whether we have reached some upper bound of what is possible in the continual learning scenario we are operating in or whether the gap could be closed by further fine-tuning the generative replay implementation. In a simple experiment, which was also performed before in [44], we want to see whether this gap in accuracy has something to do with the learning setup or whether it is a limitation of the generator. The latter is a central part of the algorithm and it is known to be a difficult task to generate high quality images. In the experiments we take the scholar that was trained on the entire MNIST data set (which we will call experience zero) and in every subsequent experience we continue the training on a data set purely consisting of replay data created by the generator from the previous experience. The replay data sets that are newly generated for each experience contain the same number of data points as the original MNIST data set. Unlike in [44] we find that running GR "over itself" does not preserve the original accuracy. Instead, after the first experience trained solely on replay data the accuracy takes a dip. It continues to decrease over the following 32 experiences we have run, although it does so at a decreasing rate of change (see Figure 9). We can only assume that the average accuracy would eventually converge to an equilibrium where the replays are good enough to maintain a certain accuracy, but due to limited computing capacity we were unable to let the experiment run for longer. Interestingly, while the class accuracies for digits five and nine are pulling the overall average down, when looking at the replay samples, it is digits one and seven that have become unrecognizable to the human eye (see Figure 10). So even from blurry images the solver is able to obtain enough information on the underlying structure of the original data set. In fact, after 33 experiences we still record an average accuracy of 79.73%, which is in the range of results we have been able to achieve in our splitMNIST experiments.

If we randomly reinitialize the classifier model's parameters before each experience and run the experiment again, we find that the initial dip is larger but the training then stabilizes in a similar way albeit less smoothly. Furthermore, the accuracies are roughly 10% below the counterparts of where we kept the weights of the classifier when continuing the training (see Figure 9 for the accuracies). This shows that most of the information about the past that is encoded in the weights of the network, can be relearned from the replay data. Nonetheless, it is highly recommendable to reuse the weights of the network in consecutive experiences when training continually. As [49] put it: it's easier to remember than to learn new knowledge. The above evidence suggests that the generator's ability acts as the upper bound for the generative replay approach instead of any characteristics inherent to the continual learning setup itself. It should therefore prove worthwhile to employ and fine-tune a more powerful generative model. In fact, [28] reports an accuracy of 95.81% for GR on splitMNIST using a GAN compared to 79.38% using a VAE with otherwise equal conditions. These and our findings support the notion that some of the difficulty of Continual Learning is shifted towards the difficulty of training good generative models.

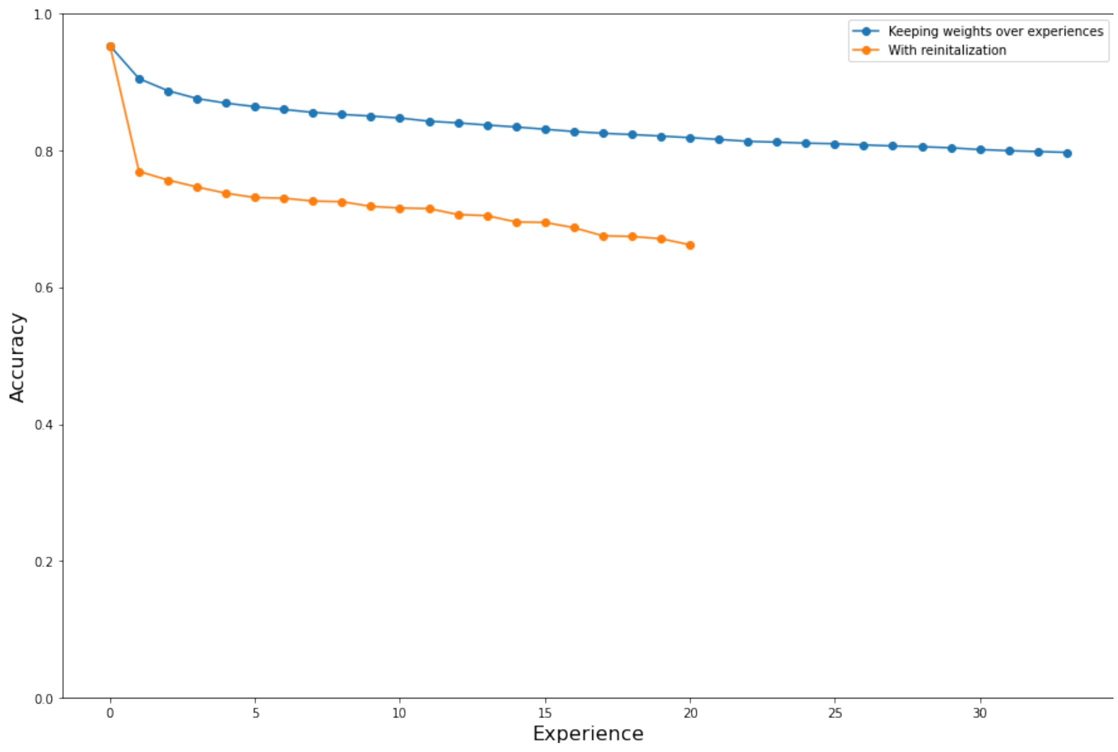

Figure 9: Accuracies for continuing training a trained scholar repeatedly solely on replay data generated by the generator of the scholar. In one case the model is kept as it is, in the other one the weights of the classifier model are reinitialized before each experience. Experience 0 corresponds to a joint training on the entire MNIST data set.

6 Experiments: Novel, Non-stationary Streaming Scenarios

Several benchmarks are established in the continual learning community which are summoned upon when evaluating a newly proposed algorithm trying to prove its superiority. In the class-incremental setup, an example of such a benchmark is splitMNIST, which is the one we have been using in Chapter 5. Similarly, splitCIFAR100 is constructed in the same way but uses a more complex and challenging data set. New data sets have been proposed, which are specifically designed for continual learning scenarios, such as CORe50, which has a temporal correlation between the data points of each class [30]. However, criticism has been voiced recently that these benchmarks help to compare algorithms in a theoretical setup but they are lacking the generality to predict their success in real-life applications. Aljundi et al. argue that "the methods developed [...] all too often depend on knowing the task boundaries. These boundaries indicate good moments to consolidate knowledge, namely after learning a task. Moreover, data can be shuffled within a task so as to guarantee independent and identically distributed (i.i.d.) data. In an online setting, on the other hand, data needs to be processed in a streaming fashion and data distributions might change gradually" [1]. As a first consequence, this deems the task-IL and domain-IL scenarios as too unrealistic for wider application, which is why we are only studying the more challenging class-IL setup in our experiments. The reason that many of the

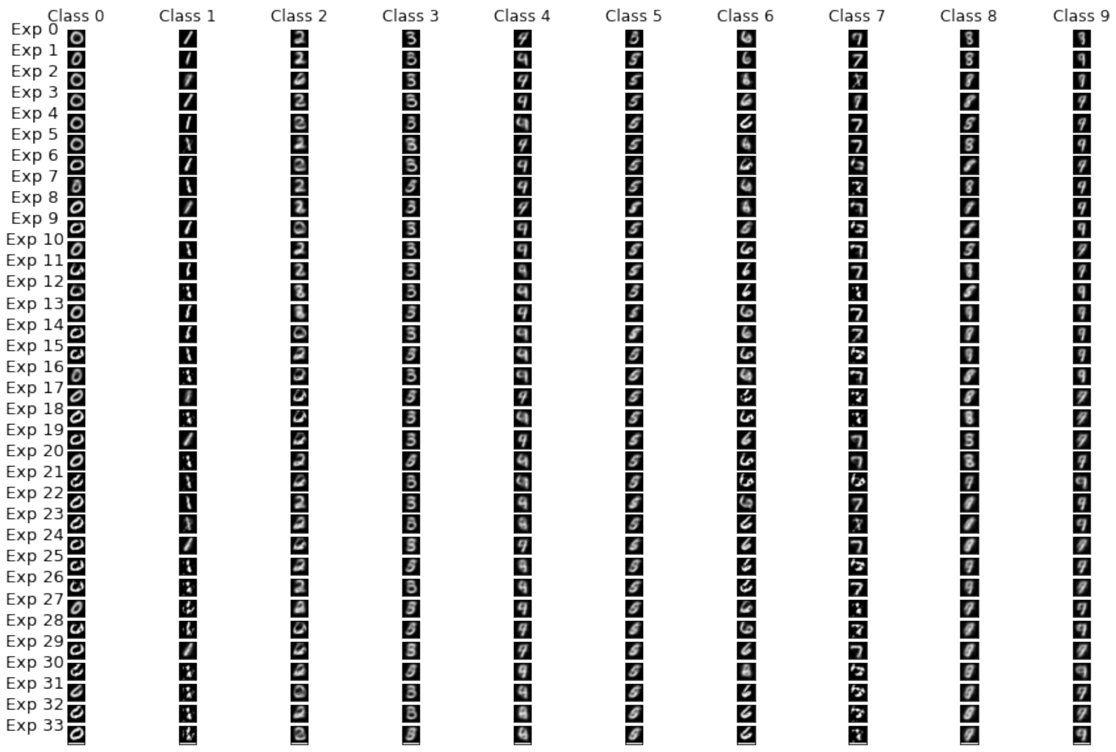

Figure 10: Examples of Replay Samples for each class (0-9) in each experience. Experience i is trained on the replay samples from experience i-1. Experience 0 was trained on the entire MNIST data set.

methods depend on the mentioned requirements is that existing benchmarks nudge researchers to design and publish algorithms that excel on those. And the existing benchmarks do not reflect real-life applications. The way Christoper Kanan puts it in a guest lecture is that: "Many are solving the wrong problem" [16, 20]. He is relating precisely to the issue that current algorithms are able to solve specific toy examples but that they are not developed to thrive in realistic settings.

Goal: Our **second goal** of this thesis and content of this chapter is the definition of scenarios that reflect a more realistic setup and the evaluation of several continual learning strategies on those scenarios. The hypothesis is that we can show that many strategies are not able to succeed under these conditions, despite being successful on the established benchmark scenarios. We therefore want to contribute to a new evaluation framework for CL methods.

We used the aforementioned criticism as a starting point and we have identified three characteristics that will likely be encountered in real-world (online) scenarios and we designed experiments to evaluate GR and other strategies from the perspective of these characteristics. The first characteristic is that the data may arrive in any order. Unlike in the joint training case, the entire data set may not be available up front and it cannot be shuffled to obtain i.i.d. batches during training, but instead a data stream might emit the data in any order and a stream training strategy should be able to handle this. Another characteristic is that the data set itself might not be uniformly distributed among the classes. This distribution could also change over time. And fi-

Order A	Order B	Order C	Order D	Order E	Average
59.25	**71.09**	68.91	61.59	67.05	65.58

Table 2: Overall accuracies in percentage (%) for the Vanilla Generative Replay algorithm on splitMNIST for 5 different orders of classes. Additionally the average over those 5 values. The orders are **A:** [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], **B:** [9, 8, 7, 6, 5, 4, 3, 2, 1, 0], **C:** [0, 8, 5, 2, 3, 6, 4, 1, 7, 9], **D:** [4, 1, 7, 5, 3, 9, 0, 8, 6, 2], **E:** [5, 4, 8, 2, 6, 9, 1, 7, 0, 3]

nally, the size of each experience might vary and could be as small as a single sample at a time.

Experiments: Our experiments will be structured in the following way: as a pre-experiment, we will **change the order** of the classes in the splitMNIST scenario to show that merely changing the order can influence the performance of a CL algorithm significantly (Section 6.1). Then, in Section 6.2, we will introduce a **new scenario** which includes training on an imbalanced data set together with a concept drift (i.e. the change of the underlying data distribution over time) as well as reappearing classes in later experiences. We will evaluate various CL approaches and see that Generative Replay outperforms the state-of-the-art method iCaRL in this scenario. Finally in Section 6.3 we evaluate different CL strategies on experiences with as few as **64 samples**, and see that generative replay fails under these conditions.

6.1 Changing the Class Order of splitMNIST

During the experiments in the previous chapter, we split the MNIST data set in numerically ascending order to make the plots more intuitive and easier to read. In the common joint training scenario, the data set is usually shuffled before training an epoch. In more realistic continual learning setups, we do not have control over the order in which the data points arrive. We want to show that this is an important consideration to make, without diving too deep into this aspect. We therefore, as a simple experiment, rerun the splitMNIST scenario for the Vanilla Generative Replay approach in the same setup as before except that we change the order in which we encounter the classes during training. And in fact, Table 2 shows that each run comes with a different accuracy. The numerically ascending order results in the lowest accuracy with 59.25%, the descending order on the other hand comes with the highest accuracy of 71.06%. The other orders were chosen at random and their respective accuracies lie in between, so that we get an average over all runs of 65.58%. Best and worst run separate more than 10 percentage points and we see that our vanilla implementation, on average, fares some 5% better than in Chapter 5, where we just looked at the ascending order.

This experiment shows that small changes in the scenario can influence the output of it. It is therefore advisable to always run a benchmarks for multiple different settings, such as with different class orders in the splitMNIST case, and compute the average as a final result. With this in mind we will now focus on the remaining two characteristics we identified above. For this, we will introduce a new kind of scenario.

6.2 Evaluation of GR and Other Methods on a New Scenario

Our aim is to create a benchmark that uses imbalanced experiences, where the distribution of the experiences changes over time and classes disappear in some experiences and reappear in others. The scenario we introduce here uses MNIST as underlying data set and is structured as follows: each experience contains images of four classes, with always one class contributing 1/10th of the entire set of class images, another one contributes 2/10th, another 3/10th and the last one 4/10th. This way, we have an imbalanced data set in each experience. In the next experience, we change the contribution of the classes in a sliding window manner, so that each class will assume each of the contribution levels once throughout training. The process of creating the scenario is also visualised in Figure 11. Through the sliding window, we simulate a concept drift within the data with some classes not appearing anymore and others reentering the training after some experiences. After ten such experiences, we have covered the entire MNIST data set and the model has seen the same amount of training data as during the regular splitMNIST scenario.

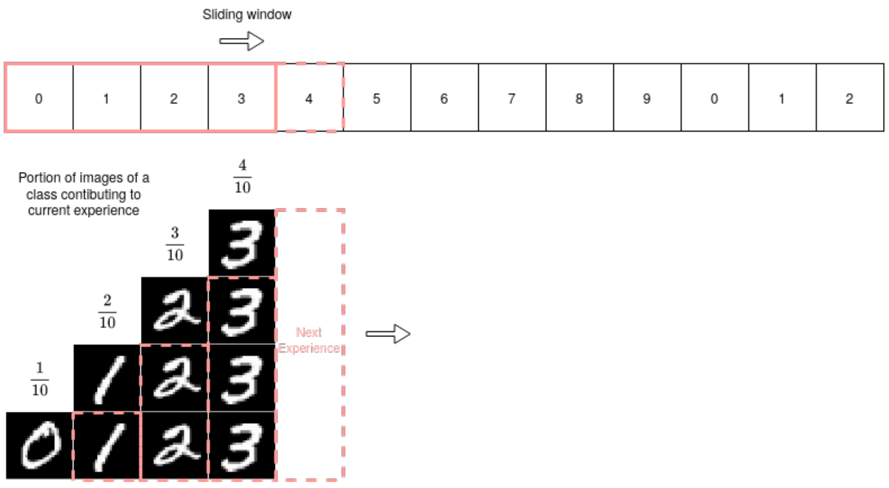
Sliding window
0
1
2
3
4
5
6
7
8
9
0
1
2
Portion of images of a class contibuting to current experience
1/10
2/10
3/10
4/10
Next Experience

Figure 11: Process of constructing the new non-stationary streaming scenario. Each experience contains images of four classes where the distribution of classes is imbalanced and changes over time: Each of the four classes contributes either 1/10, 2/10, 3/10 or 4/10 of the respective data set of class images. The next experience is constructed via a sliding window approach, such that a class that contributed 4/10 in one experience will contribute 3/10 in the next and eventually disappear from the data stream. Only classes 0, 1 and 2 reappear after not being part of an experience anymore.

We trained our scholar model from Chapter 5 on this new scenario using Generative Replay with a fixed replay size of 200. Furthermore, we evaluated and compared its performance to a range of other methods on this new scenario. As pure rehearsal strategy, we used Exact Replay, regularization methods include EWC, LwF and SI [23, 29, 51]. Hybrid methods were GEM and iCaRL [32, 39]. We also used naive training to obtain a lower bound. The resulting average accuracies for each strategy in each experience are plotted in Figure 12. Generative Replay is able to beat the iCaRL strategy, which is the state-of-the-art on common benchmarks [50]. This is because the algorithm of iCaRL is computing class exemplars after encountering a new class [39], which are then used for classification later on. The algorithm is simply not designed for the case where classes reappear in later experiences and where their exemplar would need to be updated. Regularization based methods LwC, SI and EWC fail entirely and perform just as bad as the naive training. This is not surprising as we are dealing with a class-IL scenario which has been proven a difficult challenge for these algorithms in general [50]. It has been hypothesised that some form of replay might be necessary for this group of scenarios [49]. Only Exact Replay and GEM are performing better than Generative Replay. For Exact Replay this is no surprise as Generative Replay is essentially trying to mimic Exact Replay but without the need to store actual data. GEM is performing extraordinarily well on the scenario and beating even Exact Replay.

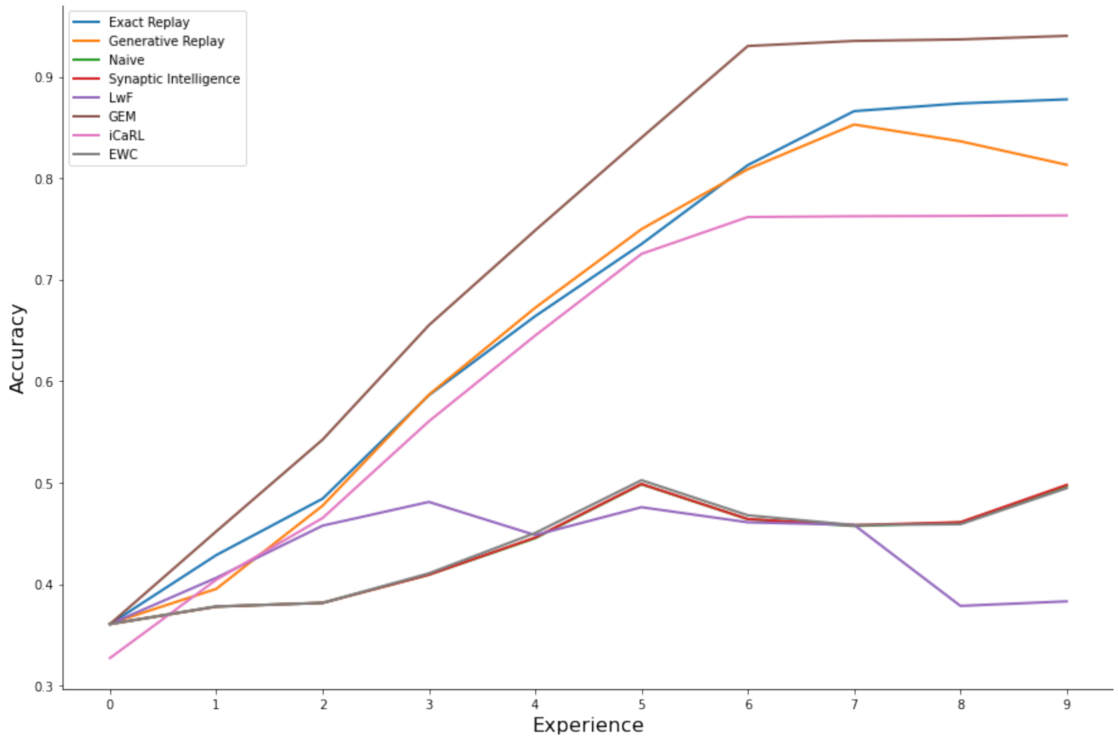

Figure 12: New imbalance scenario: Accuracy on the MNIST test set after each experience. Comparison of eight different continual learning strategies. Replay based methods were allowed a buffer of 200 replay samples. Naive training and Synaptic Intelligence have almost the same output and their accuracies are almost indistinguishable in this plot.

We would like to mention that, when designing the scenario, we found that imbalance itself is not prone to *catastrophic* forgetting as long as all classes are present. This is in line with the observation that even a few replay data points (be it for generative replay or any other rehearsal based method, such as exact replay) helps alleviating catastrophic forgetting. Therefore, we chose a scenario where we only used four classes per experience.

6.3 Reducing the Experience Size of splitMNIST

As mentioned before, a prime example of possible real life applications of continual learning are autonomous agents that interact with an environment and are required to update their model of the world in real-time and thus continually learn. In such a case the agent would be collecting single data points and there might be no time to postpone the training until many data points have been collected. Instead we would like to have a training paradigm in which we can train experiences of arbitrarily small size. In this section we will look at what happens when we train the same model as before in a class-IL scenario with only 64 images in each experience.

The scenario we use is kept simple: We split the MNIST data set into two data sets that contain the images of five classes each. We train the data sets consecutively, but instead of doing this in two experiences, we create experiences of 64 samples each, which is equivalent to training one mini-batch of data per experience. When using 16000 data points this yields $16000/64 = 252$ experiences of which the first 126 contain samples of the first five classes and the other 126 experiences contain the remaining classes. If the split into several experiences does not matter to the performance, we would expect our algorithms to solve this problem easily as it is no harder than the imbalance scenario from Section 6.2 or the splitMNIST scenario with one class per experience.

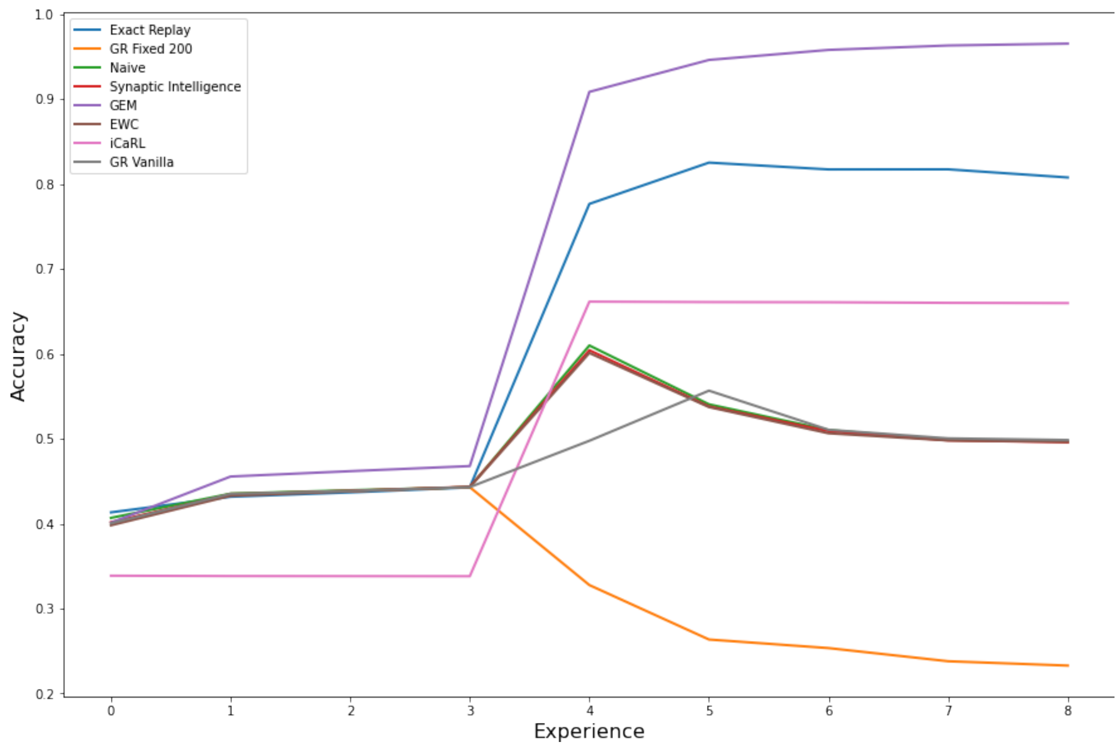

Figure 13: Experience size scenario: Accuracy on the MNIST test set after each experience. Comparison of eight different continual learning strategies. Replay-based methods were allowed a buffer of 200 replay samples. Naive training and Synaptic Intelligence have almost the same output and their accuracies are almost indistinguishable in this plot.

However, the results show that Generative Replay fails in this scenario together with the regularization-based methods. Whereas iCaRL and Exact Replay solve the problem fairly well and GEM is excelling at it, seemingly not being affected by the reduced experience size (see Figure 13). While the vanilla GR implementation shows a similar performance as the naive training, using a fixed replay size of 200 replays actually deteriorates the results further.

So why is it that Generative Replay is seemingly unable to handle experiences with few samples? One possible cause, which at the same time is a weak spot of the GR algorithm in general, is the interdependence between generator and model when producing labeled replay samples. For example, when looking at the images the generator generates together with their predicted labels, we see that generator and classifier do not adjust to the shift in the data stream in the same amount of time in the fixed replay size case. While the generator already learned to generate the new classes, the classifier was still producing old labels for them. This then causes a snowball effect where the classifier learns and consolidates the wrongly labeled data pairs, which makes it hard to exit this loop. The reason why we didn't experience this issue for the normal splitMNIST scenario is that we only start to add replay data after the first experience, such that the generator and model can use this experience as some kind of warm-up phase by the end of which we are generating good images with high confident labels. However, when the warm-up phase is only one mini batch long, as in our case here, and when there is a shift in the data stream, the generator and classifier have not enough time to adjust to this change before having to replay data. This can then lead to the vicious cycle described before. In this sense, the algorithm is not robust and needs fine-tuning depending on the scenario at hand. Making the algorithm robust for this kind of streaming scenarios could be the subject of further research.

We have a large number of experiences, therefore increasing the number of replay per experience or changing the weight of the loss does not work well and we would have to adjust the way we compute these two. Therefore we simply stick to a fixed replay size of 400 as we only want to see how GR generally performs on these scenarios instead of trying to find the highest accuracy.

6.4 Summary

We have run three experiments, each with focus on a different characteristic that we considered relevant in real-life continual learning settings. First was the random order a data stream might emit data in. We have setup a simple experiment where we changed the order of classes for the splitMNIST scenario. This was enough to show that the order in which a model encounters samples makes a difference for the training. From this we can deduct that it is desirable to at least rerun every continual learning experiment for different orders of data and compute the average over all runs to provide a more reliable result. We then turned to the idea that data streams might not provide a uniformly distributed data set and the distribution with which data is emitted can change over time. We have created a novel scenario for this and we have shown that iCaRL, as it is designed for very specific scenarios only, was outperformed by Generative Replay on this new scenario. Finally, we have constructed a simple scenario which is characterized by a small experiences size. Each experience only contained 64 data points, which is equivalent to the batch size we have used throughout our experiments. We have seen that due to its complex interplay of discriminator and generator, the GR strategy failed for this scenario. It could be subject of further research to adjust the algorithm to be more robust and to make it easier to use in any given setting. This is especially relevant in the light of the fact that Exact Replay, which tackles Catastrophic Forgetting in a very similar way as Generative Replay, performed well on both, the imbalance scenario as well as the experience size scenario. We have also seen that the GEM strategy has shown the best results and has proven to be working robustly in all of our scenarios. In fact, the algorithm was designed for being able to learn with only a single sample at a time [32].

7 Conclusion

At the outset of this thesis, we formulated several research questions revolving around the Generative Replay algorithm. We want to summarize here the steps we took to answer these questions and draw conclusions from the results of our experiments. Furthermore, we do want to point out possible drawbacks of the method we have identified and give an overview of further research that could build upon our findings.

In order to study the algorithm, an implementation of the former was needed that is flexible enough to easily extend the algorithm and swap scenarios, training data and architectures. We found that the continual learning framework Avalanche did provide implementations of many common continual learning algorithms, but was lacking one for Generative Replay. Our first goal was to implement the algorithm in such a modular way that satisfies the requirements of the Avalanche community as well as ours so that we could use it for subsequent experiments. We fulfilled this goal by contributing to the open-source project our plugin-based implementation, which can be used to apply Generative Replay to both a tuple of a discriminative and generative model as well as a generative on its own. We used it as foundation for the subsequent research on the method's behaviour and it is openly accessible and well documented for any other person to employ in their own research.

We then set out to ask a range of questions regarding the application of the algorithm. We pointed out that the original paper in which the algorithm was proposed did not provide details on how much replay data to generate or how to weigh the importance of the replay data relative to the real data over time. We designed five variations of the algorithms in terms of how much data is generated and how much the loss of the replay data contributes to the total loss. We reran the same splitMNIST scenario for each variation and compared their performance in terms of accuracy and forgetting. All variations prevent *catastrophic* forgetting, which occurs when naively training the model with no additional strategy at all and which we verified to result in a 10% accuracy on the MNIST test data set. This is equivalent to random guessing. We found that replaying the same amount of replay data as there is real data results in an accuracy of roughly 60% and we find that by the end of training at least the earliest classes are still forgotten. We showed that this can be improved by continually increasing the amount of replay data with each experience as well as by increasing the importance of the replay loss when calculating the total loss. This helps alleviating forgetting of entire classes completely and we reach an average accuracy of above 80%. We also showed that similar results can be obtained by enforcing a class-balanced replay data set, which can be achieved by employing a conditional generative model, which allows to generate samples of a specified class. There are many more possible variations of the algorithm and from our results, we see that it is worthwhile to study those as they greatly differ in accuracy and some might not be applicable for every scenario. For example, in scenarios with many classes and many experiences the variation of continually increasing the amount of replay data does not scale well as we quickly have to handle a huge pool of replay data. Due to its low overhead and high accuracy, we conclude that using a conditional generator constitutes the best strategy among the ones we tested.

Since the accuracies we achieved were still far below the 95% that we reach when training the model on the combined MNIST data set instead of learning class by class, we hypothesised that the generator acts as a bottleneck for the knowledge that is stored and which then leads to an upper bound on the accuracy of the discriminator that trains on that knowledge. To examine this we ran an experiment where we trained a model and a generator on the entire MNIST data set and afterwards repeatedly retrained both only on synthetic replay data produced by the generator. We did this once by reinitializing the model's weight before each retraining phase and once by keeping the weights. In the latter case we slowly converged for an accuracy of about 80% which is roughly equivalent to what we have been able to achieve in the continual learning setup. By dropping the weights and having the model learn everything from scratch, we fared about 10% worse. From this we take that most of the knowledge about the data distribution is also stored in the generator and only some additional information is encoded in the model's weights. This suggests that the performance of the generative replay method is highly dependent on the generator's performance. But since generating high dimensional data samples is a difficult task on its own, as other works have pointed out, at least part of the difficulty of training a model continually is shifted towards the equally difficult task of data generation. This complexity of the algorithm is a clear drawback of the method. Other works have suggested to generate low dimensional replay samples and to replay them along with the low dimensional representations of the input data in some hidden layer. This would greatly reduce the complexity of the data generation and our results suggest that this is a desirable goal.

Another main issue we wanted to address in our work is that the established benchmarks, such as splitMNIST, do not reflect real-world continual learning settings, as they would be encountered for example by an autonomous agent that continuously needs to adapt to changing environments. Our hypothesis was that if we define new scenarios that use more realistic assumptions, Generative Replay would succeed on them and be on par with Exact Replay and outperform other methods. We then constructed a new scenario in which the data would be imbalanced in each experience and, most crucially, classes seen before could reappear in later experiences. By evaluating a range of methods on this scenario we found that in fact Generative Replay performed almost as well as Exact Replay and it outperformed all regularization-based methods and iCaRL. The latter is impressive and at first surprising, as iCaRL is considered the state-of-the-art strategy on the common benchmarks. When looking more closely, we identified the reason of iCaRL's poor result: the algorithm is designed in such a way that it expects to encounter a class only in a single experience during the training. On the other hand, the GEM strategy, outperformed all other methods. In this sense, our experiment was very successful as we were able to identify the weaknesses of the established benchmarks and provide a new scenario on which existing continual learning strategies show a differing performance than on existing benchmarks.

We then ran a last experiment in which we decreased the experience size to a single batch of samples per experience. Otherwise we kept the order of samples unchanged as when training on splitMNIST. We believe it is important for a continual learning strategy to be able to update a model with any amount of data that becomes available at a given moment. We expected for the strategies to perform similar to how they performed on splitMNIST. The actual results showed that this hypothesis did not hold. In fact, one of the two variations of Generative Replay we ran performed so poorly that it came close to random guessing. We identified as a major reason for this problem that because the experience sizes are so small the generator has no "warm-up" phase. It is supposed to generate new samples of classes immediately after they become available. At the same time, the labels for these samples are produced by forwarding them through the model, which at the beginning is not yet properly trained either. We therefore use wrongly labeled replay samples of poor quality when continuing the training. The two characteristics of the reliance on the generator and the interdependence between generator and discriminator when producing the replay data set are major drawbacks of the Generative Replay method. Which make it difficult to control and which require knowledge of the underlying scenario to properly adjust the algorithm, for example by providing a warm-up phase for the generator. We therefore conclude that Generative Replay is not as effective, especially in more realistic setups, as we previously assumed. Furthermore, we have seen that the GEM method again performed extremely well out of the box with no adjustments needed. In fact, GEM was designed with these more realistic conditions in mind such as small experience sizes.

Our research has shown that it is important for the continual learning community to consider new benchmarks that aim at making them an indicator for whether a strategy can succeed in a real-world application. We have provided two examples of such but future research could extend those and use more complex data sets as well as different data types. Most of the related work is dealing with image data only and we believe it is important to develop strategies that work in other domains, too. From our findings, GEM has proven to be very promising whereas Generative Replay did not live up to the expectations in the more realistic scenarios. Many promising extensions have been proposed, such as using replay of lower dimensional representation or using a conditional generator in order to enforce a balanced replay data set. Future research could bring these ideas together and run them on our or other newly conceived scenarios.

References

[1] Rahaf Aljundi, Klaas Kelchtermans, and Tinne Tuytelaars. *Task-Free Continual Learning*. 2019. arXiv: 1812.03596 [cs.CV].

[2] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. DOI: 10.48550/ARXIV.2005.14165. URL: https://arxiv.org/abs/2005.14165.

[3] Angelo Cangelosi and Matthew Schlesinger. *Developmental Robotics: From Babies to Robots*. Jan. 2015. ISBN: 9780262028011. DOI: 10.7551/mitpress/9320.001.0001.

[4] Antonio Carta et al. *Ex-Model: Continual Learning from a Stream of Trained Models*. 2021. DOI: 10.48550/ARXIV.2112.06511. URL: https://arxiv.org/abs/2112.06511.

[5] Arslan Chaudhry et al. "Efficient Lifelong Learning with A-GEM". In: *CoRR* abs/1812.00420 (2018). arXiv: 1812.00420. URL: http://arxiv.org/abs/1812.00420.

[6] Zhiyuan Chen and Bing Liu. *Lifelong Machine Learning, Second Edition*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2018. DOI: 10.2200/S00832ED1V01Y201802AIM037. URL: https://doi.org/10.2200/S00832ED1V01Y201802AIM037.

[7] *ContinualAI / avalanche : Generative Replay 931*. https://github.com/ContinualAI/avalanche/pull/931. Accessed: 2022-06-22.

[8] Corinna Cortes et al. "AdaNet: Adaptive Structural Learning of Artificial Neural Networks". In: *CoRR* abs/1607.01097 (2016). arXiv: 1607.01097. URL: http://arxiv.org/abs/1607.01097.

[9] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4 (Dec. 1989), pp. 303–314. ISSN: 0932-4194. DOI: 10.1007/BF02551274. URL: http://dx.doi.org/10.1007/BF02551274.

[10] Matthias Delange et al. "A continual learning survey: Defying forgetting in classification tasks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), pp. 1–1. DOI: 10.1109/TPAMI.2021.3057446.

[11] Peter Eriksson et al. "Neurogenesis in the Adult Human Hippocampus". In: *Nature medicine* 4 (Dec. 1998), pp. 1313–7. DOI: 10.1038/3305.

[12] Robert French. "Catastrophic forgetting in connectionist networks". In: *Trends in cognitive sciences* 3 (May 1999), pp. 128–135. DOI: 10.1016/S1364-6613(99)01294-2.

[13] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. DOI: 10.48550/ARXIV.1406.2661. URL: https://arxiv.org/abs/1406.2661.

[14] *GPT-3 is No Longer the Only Game in Town*. https://lastweekin.ai/p/gpt-3-is-no-longer-the-only-game?s=r. Accessed: 2022-04-19.

[15] Stephen Grossberg. "How Does a Brain Build a Cognitive Code?" In: *Studies of Mind and Brain: Neural Principles of Learning, Perception, Development, Cognition, and Motor Control*. Dordrecht: Springer Netherlands, 1982, pp. 1–52. ISBN: 978-94-009-7758-7. DOI: 10.1007/978-94-009-7758-7_1. URL: https://doi.org/10.1007/978-94-009-7758-7_1.

[16] Tyler L. Hayes et al. *REMIND Your Neural Network to Prevent Catastrophic Forgetting*. 2020. arXiv: 1910.02509 [cs.LG].

[17] Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. New York: Wiley, June 1949. ISBN: 0-8058-4300-0.

[18] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[19] *Inputs to the forward- and criterion-functions in the BaseStrategy-object are limited 596*. https://github.com/ContinualAI/avalanche/issues/596. Accessed: 2022-06-22.

[20] *Invited Talk "Rethinking Continual Learning: How to Define Success" by Christopher Kanan*. https://www.youtube.com/watch?v=N7XJ-QTEoHI. Accessed: 2021-12-26.

[21] *Jupyter Notebooks for Generative Replay experiments*. https://github.com/travela/continual-learning/tree/master/notebooks. Accessed: 2022-06-22.

[22] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2013. DOI: 10.48550/ARXIV.1312.6114. URL: https://arxiv.org/abs/1312.6114.

[23] James Kirkpatrick et al. “Overcoming catastrophic forgetting in neural networks”. In: *CoRR* abs/1612.00796 (2016). arXiv: 1612.00796. URL: http://arxiv.org/abs/1612.00796.

[24] Jeremias Knoblauch, Hisham Husain, and Tom Diethe. “Optimal Continual Learning Has Perfect Memory and is NP-HARD”. In: *Proceedings of the 37th International Conference on Machine Learning*. ICML’20. JMLR.org, 2020.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), p. 436.

[27] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[28] Timothée Lesort et al. “Generative Models from the perspective of Continual Learning”. In: *2019 International Joint Conference on Neural Networks (IJCNN)*. 2019, pp. 1–8. DOI: 10.1109/IJCNN.2019.8851986.

[29] Zhizhong Li and Derek Hoiem. “Learning without Forgetting”. In: *CoRR* abs/1606.09282 (2016). arXiv: 1606.09282. URL: http://arxiv.org/abs/1606.09282.

[30] Vincenzo Lomonaco and Davide Maltoni. “CORe50: a New Dataset and Benchmark for Continuous Object Recognition”. In: *Proceedings of the 1st Annual Conference on Robot Learning*. Ed. by Sergey Levine, Vincent Vanhoucke, and Ken Goldberg. Vol. 78. Proceedings of Machine Learning Research. PMLR, 13–15 Nov 2017, pp. 17–26. URL: https://proceedings.mlr.press/v78/lomonaco17a.html.

[31] Vincenzo Lomonaco et al. "Avalanche: an End-to-End Library for Continual Learning". In: *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. 2nd Continual Learning in Computer Vision Workshop. 2021.

[32] David Lopez-Paz and Marc'Aurelio Ranzato. "Gradient Episodic Memory for Continuum Learning". In: *CoRR* abs/1706.08840 (2017). arXiv: 1706.08840. URL: http://arxiv.org/abs/1706.08840.

[33] James Mcclelland, Bruce Mcnaughton, and Randall O'Reilly. "Why There are Complementary Learning Systems in the Hippocampus and Neocortex: Insights from the Successes and Failures of Connectionist Models of Learning and Memory". In: *Psychological review* 102 (Aug. 1995), pp. 419–57. DOI: 10.1037/0033-295X.102.3.419.

[34] Kenneth Miller and David MacKay. "The Role of Constraints in Hebbian Learning". In: *Neural Comput.* 6 (July 1997). DOI: 10.1162/neco.1994.6.1.100.

[35] *OpenAI's massive GPT-3 model is impressive, but size isn't everything*. https://venturebeat.com/2020/06/01/ai-machine-learning-openai-gpt-3-size-isnt-everything/. Accessed: 2022-04-19.

[36] German I. Parisi et al. "Continual lifelong learning with neural networks: A review". In: *Neural Networks* 113 (2019), pp. 54–71. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2019.01.012. URL: https://www.sciencedirect.com/science/article/pii/S0893608019300231.

[37] Dushyant Rao et al. "Continual Unsupervised Representation Learning". In: *CoRR* abs/1910.14481 (2019). arXiv: 1910.14481. URL: http://arxiv.org/abs/1910.14481.

[38] O'Reilly RC and Rudy JW. "Computational principles of learning in the neocortex and hippocampus". In: *Hippocampus* 10 (4 2000), pp. 389–397. DOI: doi:10.1002/1098-1063(2000)10:4<389::AID-HIPO5>3.0.CO;2-P.

[39] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, and Christoph H. Lampert. "iCaRL: Incremental Classifier and Representation Learning". In: (2016). arXiv: 1611.07725. URL: http://arxiv.org/abs/1611.07725.

[40] Anthony Robins. "Catastrophic forgetting in neural networks: the role of rehearsal mechanisms". In: *Proceedings 1993 The First New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems*. 1993, pp. 65–68. DOI: 10.1109/ANNES.1993.323080.

[41] Anthony Robins. "Catastrophic Forgetting, Rehearsal and Pseudorehearsal". In: *Connection Science* 7.2 (1995), pp. 123–146. DOI: 10.1080/09540099550039318. eprint: https://doi.org/10.1080/09540099550039318. URL: https://doi.org/10.1080/09540099550039318.

[42] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *CoRR* abs/1409.0575 (2014). arXiv: 1409.0575. URL: http://arxiv.org/abs/1409.0575.

[43] Andrei A. Rusu et al. "Progressive Neural Networks". In: *CoRR* abs/1606.04671 (2016). arXiv: 1606.04671. URL: http://arxiv.org/abs/1606.04671.

[44] Hanul Shin et al. *Continual Learning with Deep Generative Replay*. 2017. arXiv: 1705.08690 [cs.AI].

[45] Konstantin Shmelkov, Cordelia Schmid, and Karteek Alahari. “Incremental Learning of Object Detectors without Catastrophic Forgetting”. In: *CoRR* abs/1708.06977 (2017). arXiv: 1708.06977. URL: http://arxiv.org/abs/1708.06977.

[46] David Silver et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search”. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[47] *The GPT-3 economy*. https://bdtechtalks.com/2020/09/21/gpt-3-economy-business-model/. Accessed: 2022-04-19.

[48] *Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, the World’s Largest and Most Powerful Generative Language Model*. https://www.microsoft.com/en-us/research/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/. Accessed: 2022-04-19.

[49] G.M. van de Ven, H.T. Siegelmann, and A.S. Tolias. “Brain-inspired replay for continual learning with artificial neural networks”. In: *Nat Commun* 11 (2020), p. 4069. ISSN: 0893-6080. DOI: https://doi.org/10.1038/s41467-020-17866-2.

[50] Gido M. van de Ven and Andreas S. Tolias. *Three scenarios for continual learning*. 2019. arXiv: 1904.07734 [cs.LG].

[51] Friedemann Zenke, Ben Poole, and Surya Ganguli. “Improved multitask learning through synaptic intelligence”. In: *CoRR* abs/1703.04200 (2017). arXiv: 1703.04200. URL: http://arxiv.org/abs/1703.04200.

A Appendix

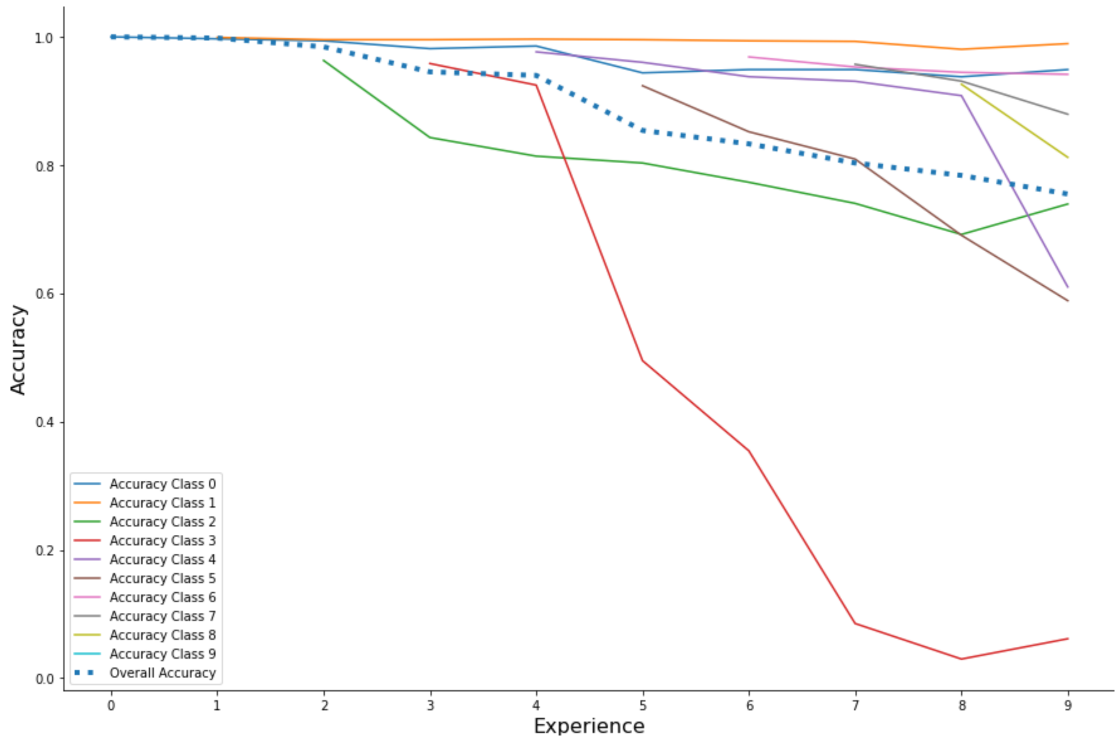

Figure 14: **Fixed Replay Size:** Results of the GR implementation using 400 replay images in each training iteration. Accuracies for each experience of the splitMNIST scenario, where each experience shows the accuracy for every class that has been encountered until that point plus the average over those classes.

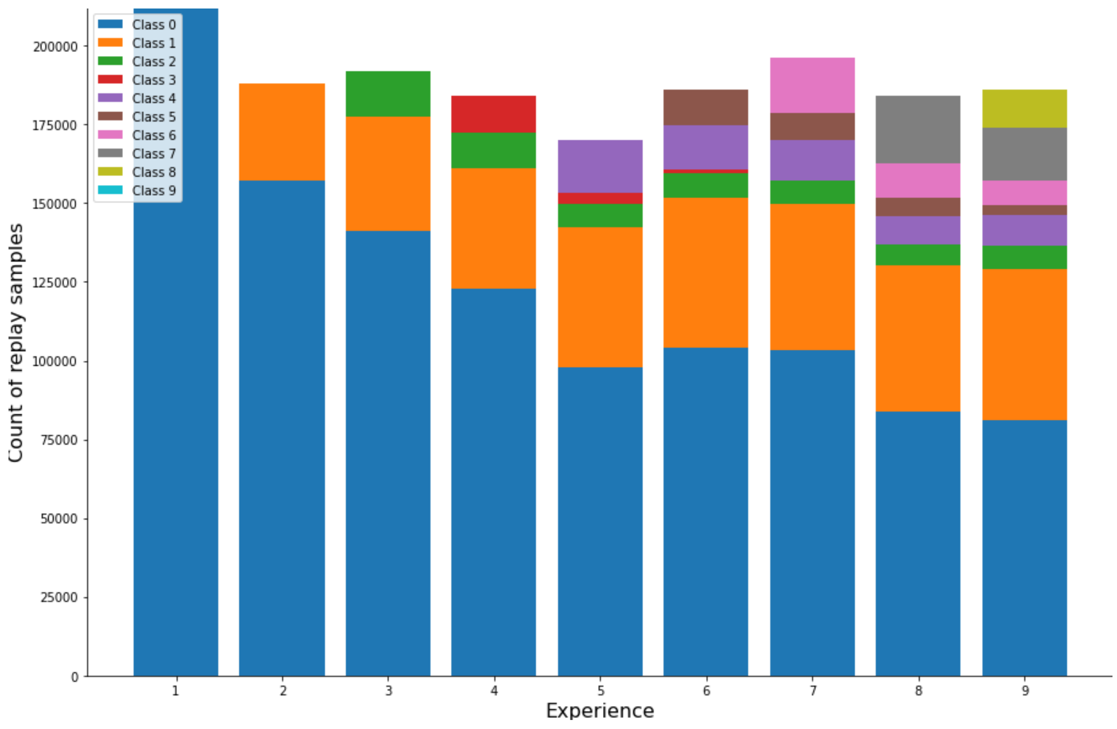

Figure 15: **Fixed Replay Size:** Distribution of replay labels among the ten classes. Plotted in a single bar for each experience (starting from experience 1; there are no replay samples used in experience 0).

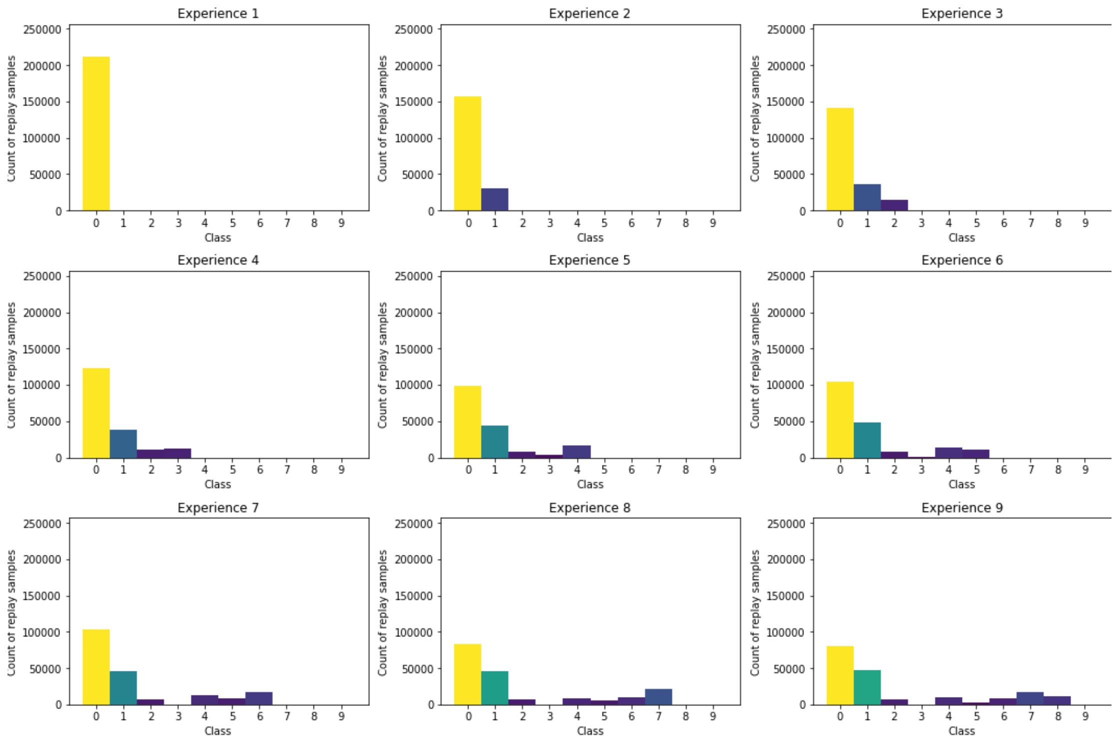

Figure 16: **Fixed Replay Size:** Distribution of replay labels among the ten classes. Plotted in a separate bar plot for each experience (starting from experience 1; there are no replay samples used in experience 0).

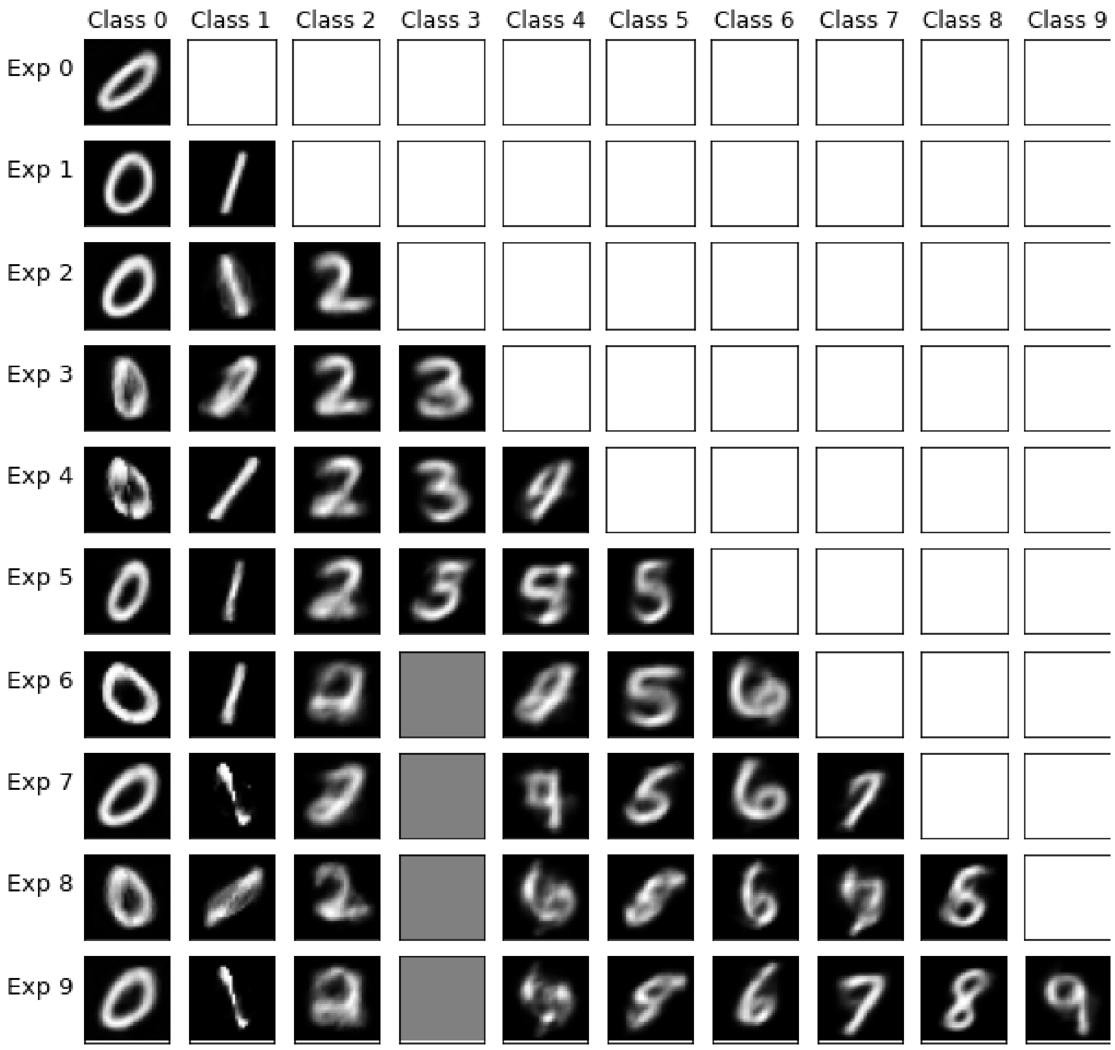

Figure 17: **Fixed Replay Size:** Samples of images generated for each class after training an experience. White tiles mean that this class has not been encountered yet, grey tiles indicate that a class has been forgotten.

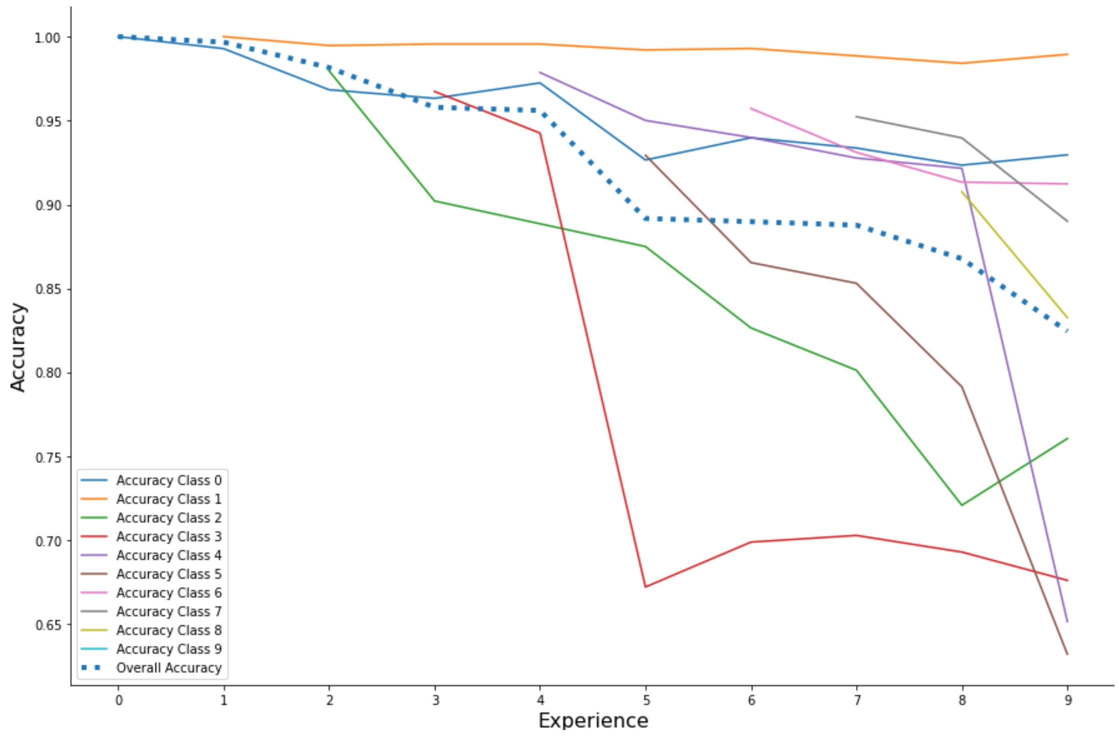

Figure 18: **Increasing number of replay samples:** Results of the GR implementation increasing the number of replay samples in each experience by 64 (batch size). Accuracies for each experience of the splitMNIST scenario, where each experience shows the accuracy for every class that has been encountered until that point plus the average over those classes.

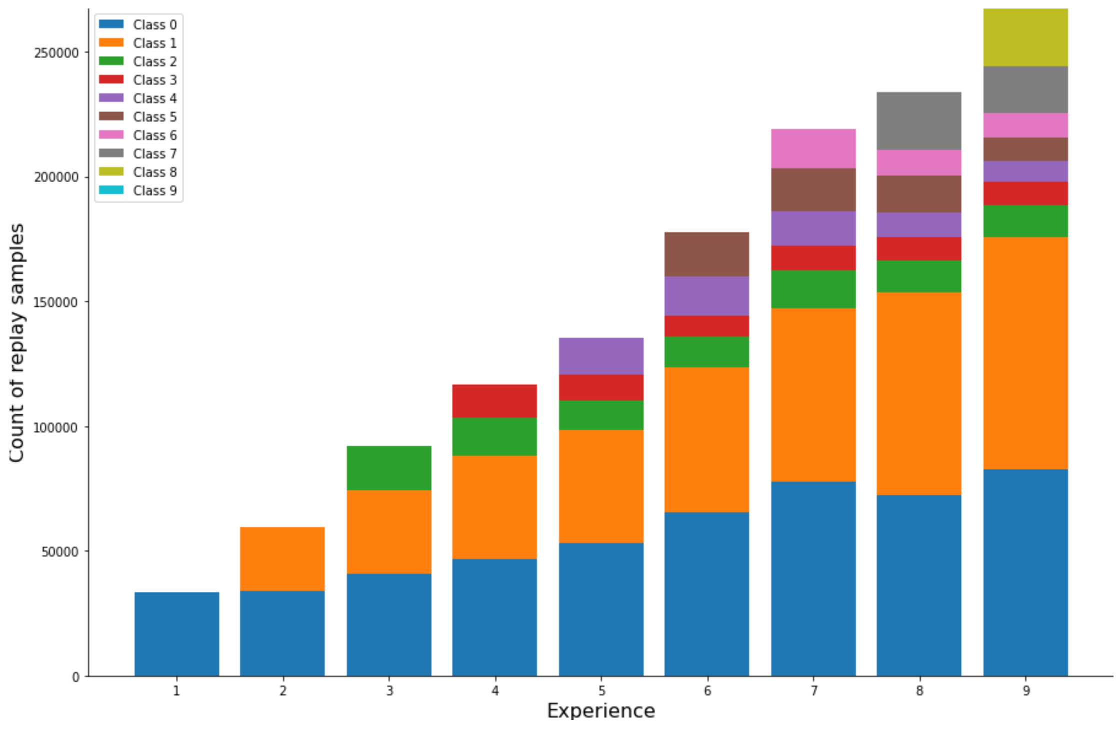

Figure 19: **Increasing number of replay samples:** Distribution of replay labels among the ten classes. Plotted in a single bar for each experience (starting from experience 1; there are no replay samples used in experience 0).

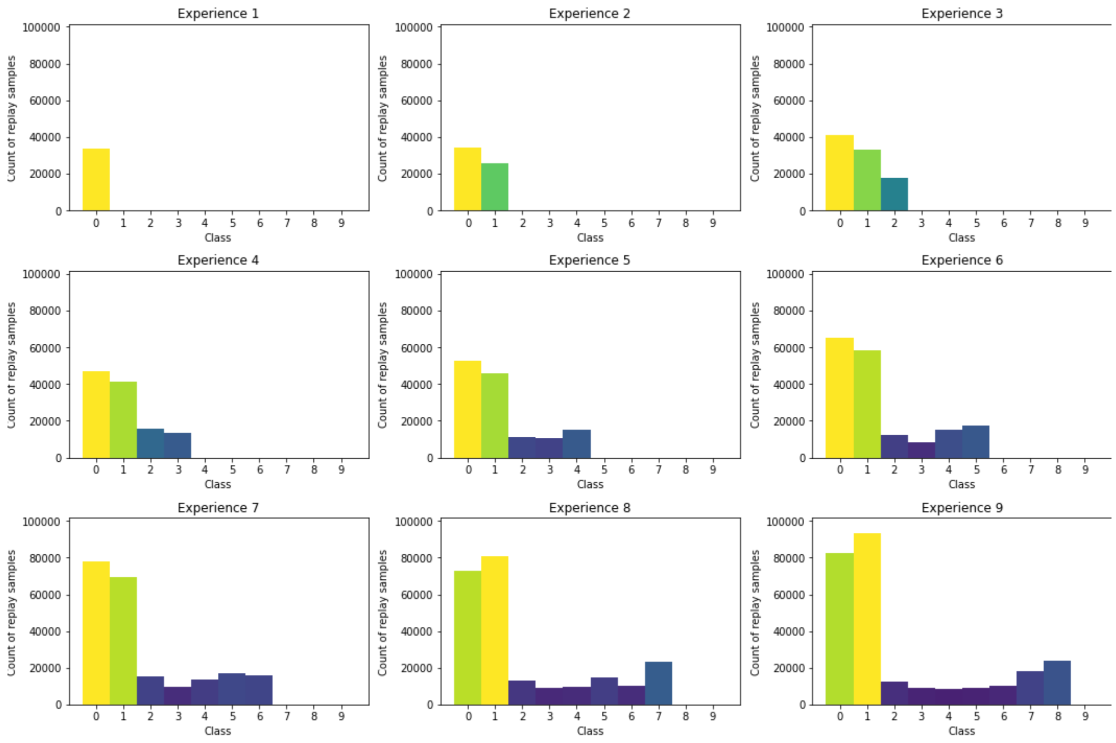

Figure 20: **Increasing number of replay samples:** Distribution of replay labels among the ten classes. Plotted in a separate bar plot for each experience (starting from experience 1; there are no replay samples used in experience 0).

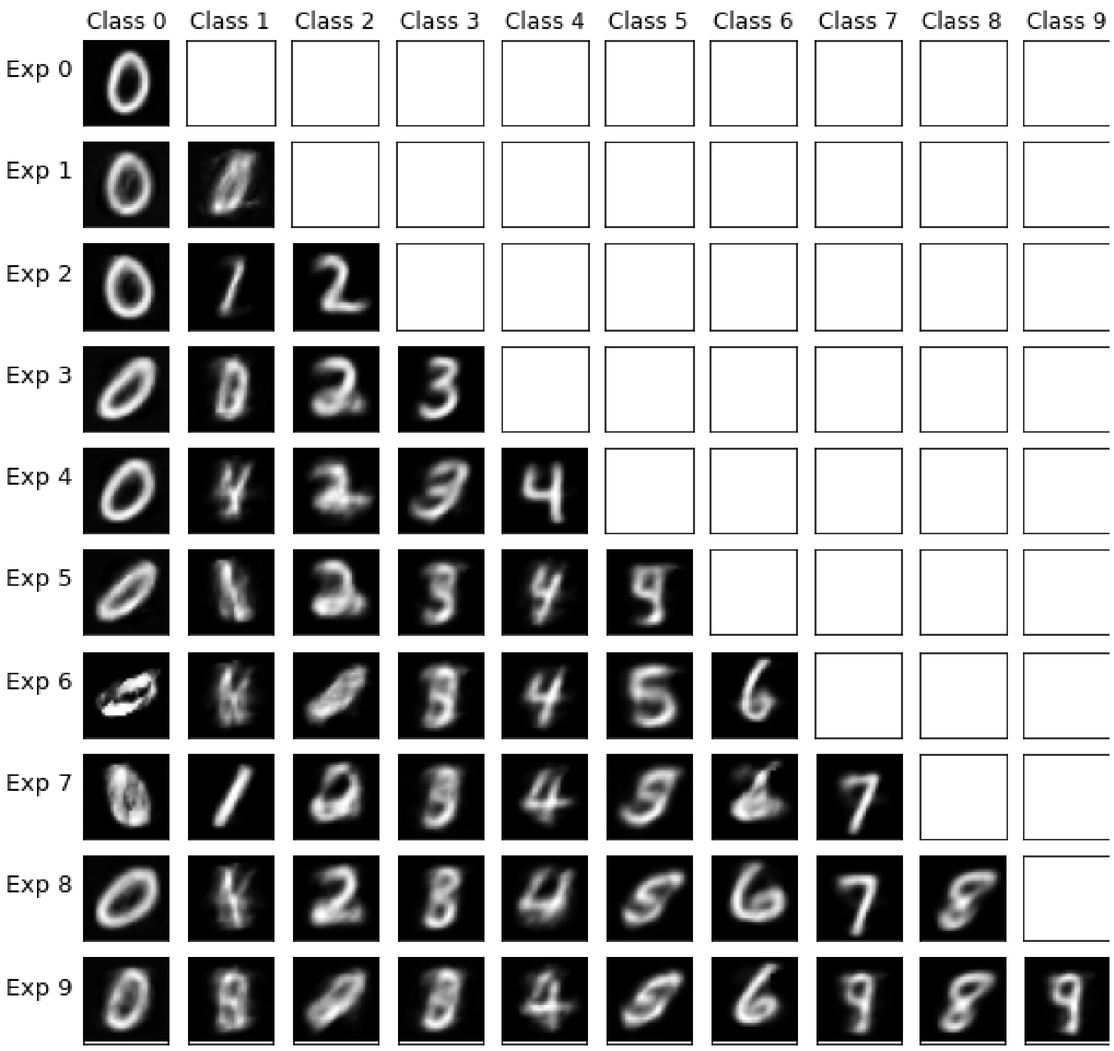

Figure 21: **Increasing number of replay samples:** Samples of images generated for each class after training an experience. White tiles mean that this class has not been encountered yet, grey tiles indicate that a class has been forgotten.

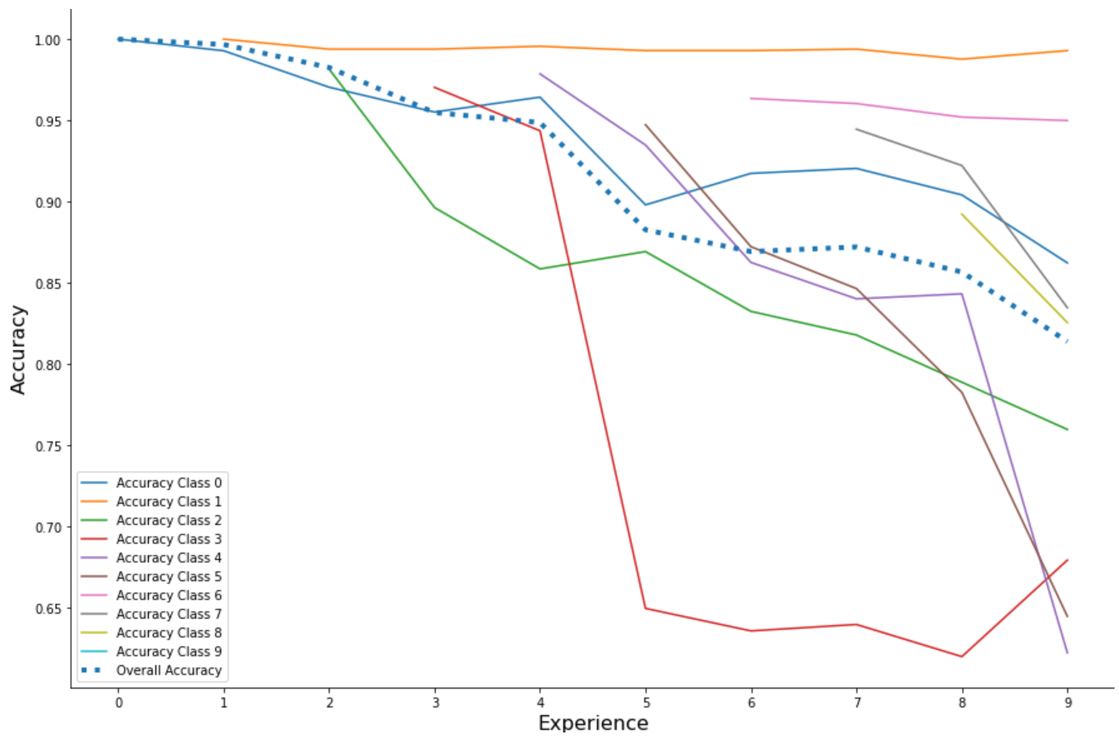

Figure 22: **Weighted Loss Function:** Results of the GR implementation using a weighted loss term. Accuracies for each experience of the splitMNIST scenario, where each experience shows the accuracy for every class that has been encountered until that point plus the average over those classes.

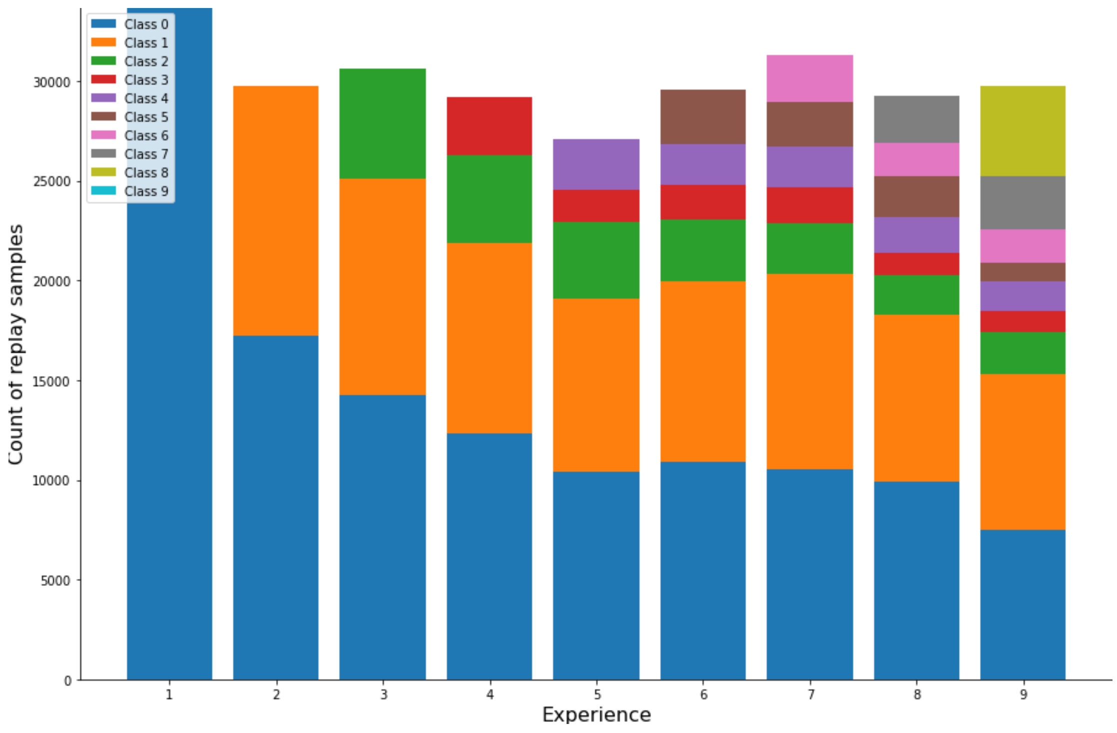

Figure 23: **Weighted Loss Function:** Distribution of replay labels among the ten classes. Plotted in a single bar for each experience (starting from experience 1; there are no replay samples used in experience 0).

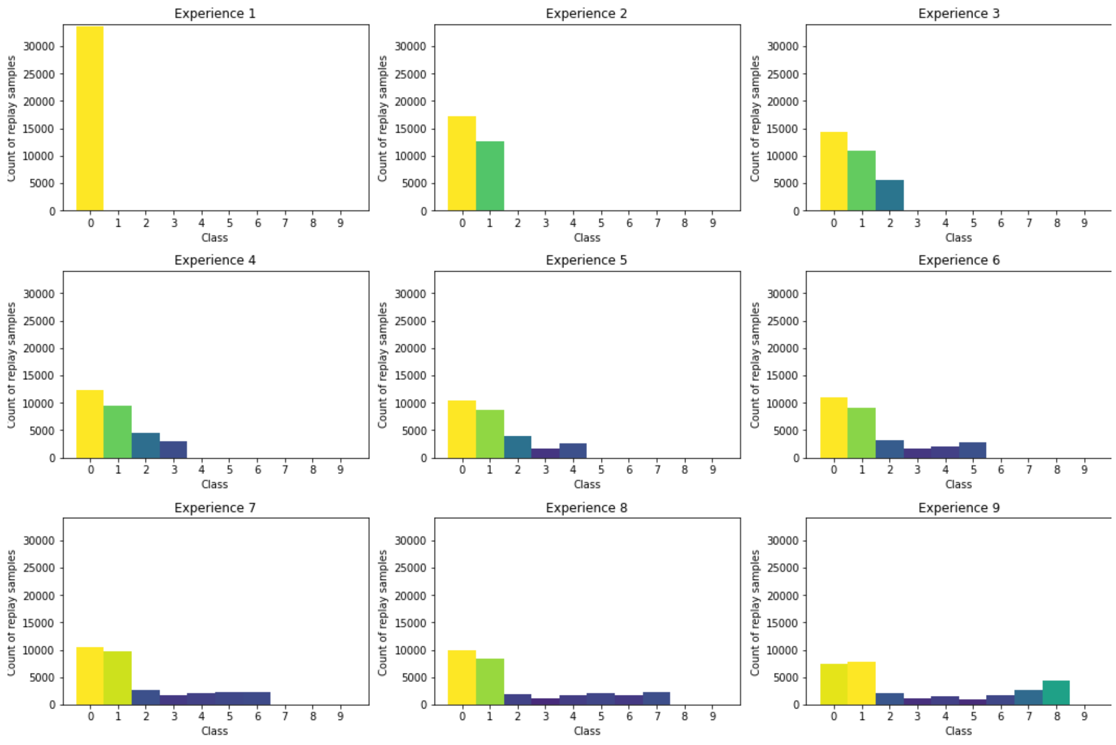

Figure 24: **Weighted Loss Function:** Distribution of replay labels among the ten classes. Plotted in a separate bar plot for each experience (starting from experience 1; there are no replay samples used in experience 0).

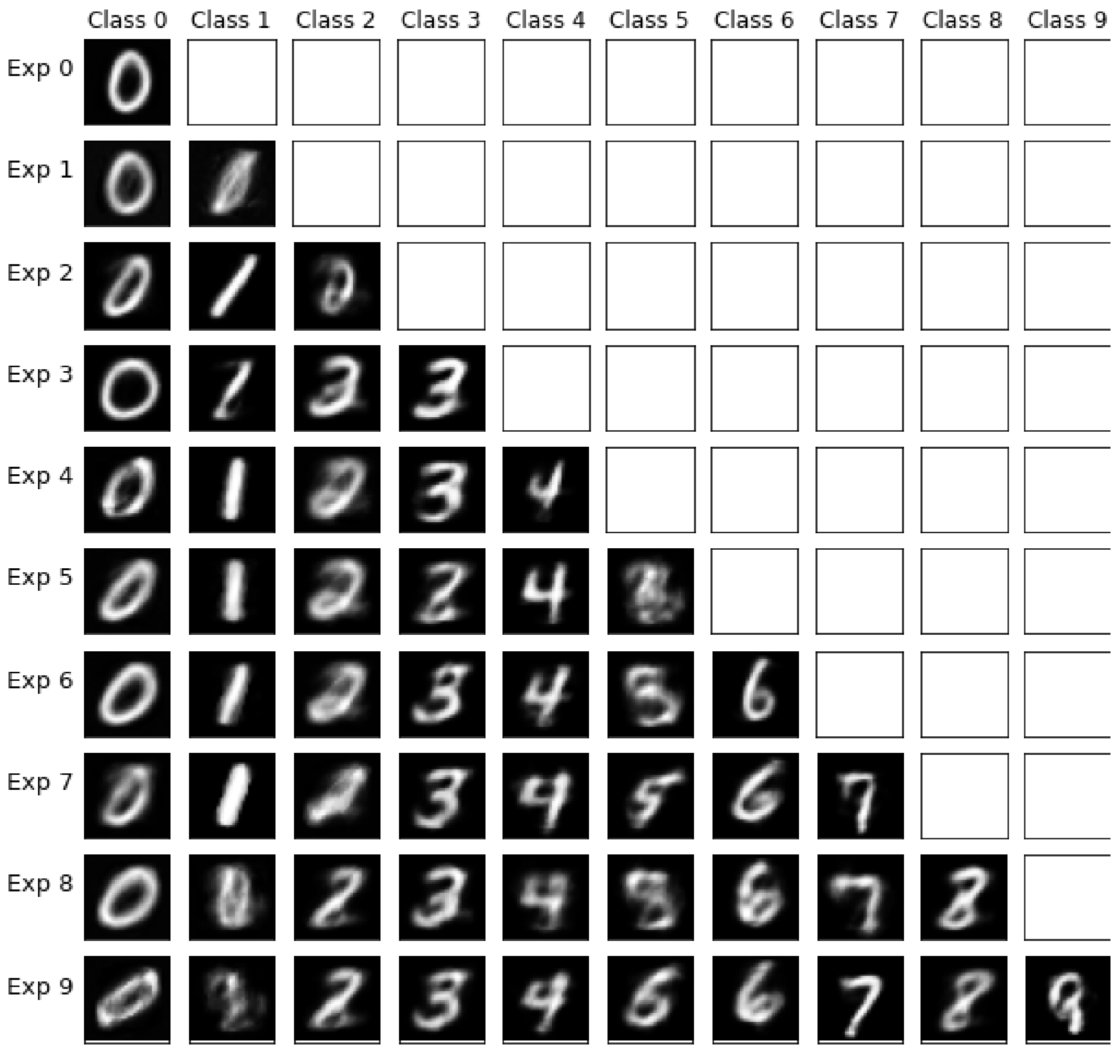

Figure 25: **Weighted Loss Function:** Samples of images generated for each class after training an experience. White tiles mean that this class has not been encountered yet, grey tiles indicate that a class has been forgotten.

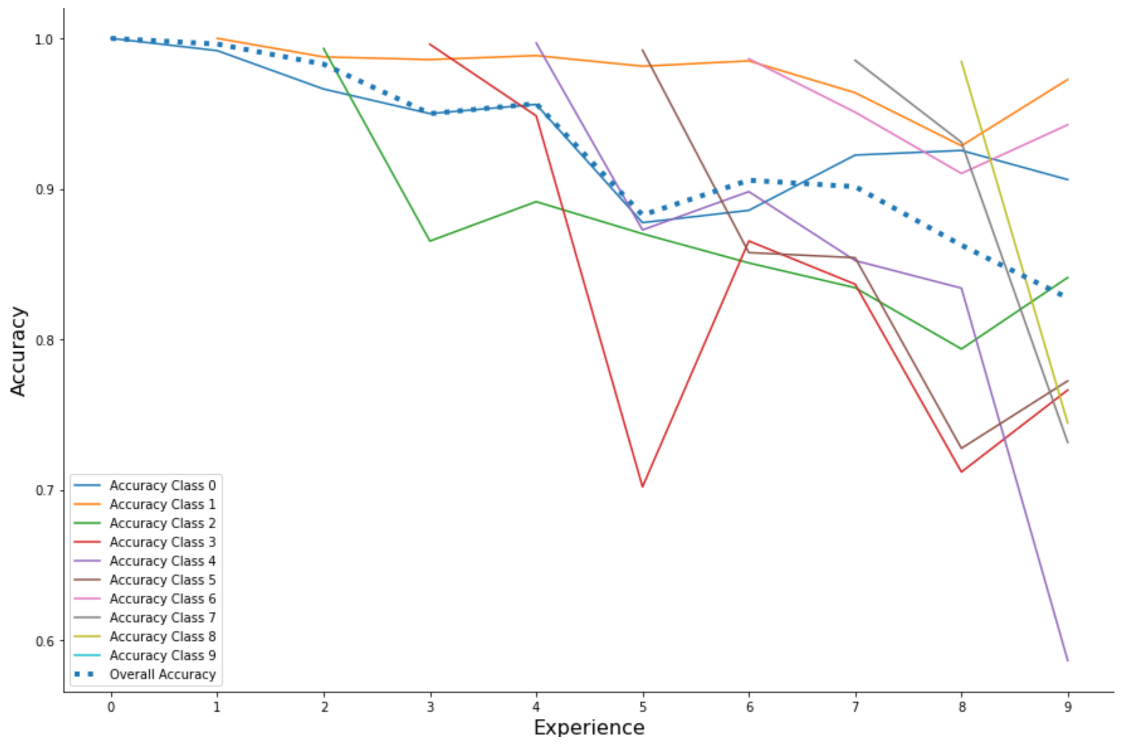

Figure 26: **Balanced Replay Sample Distribution:** Results of the GR implementation using a uniformly distributed replay data set. Accuracies for each experience of the splitMNIST scenario, where each experience shows the accuracy for every class that has been encountered until that point plus the average over those classes.

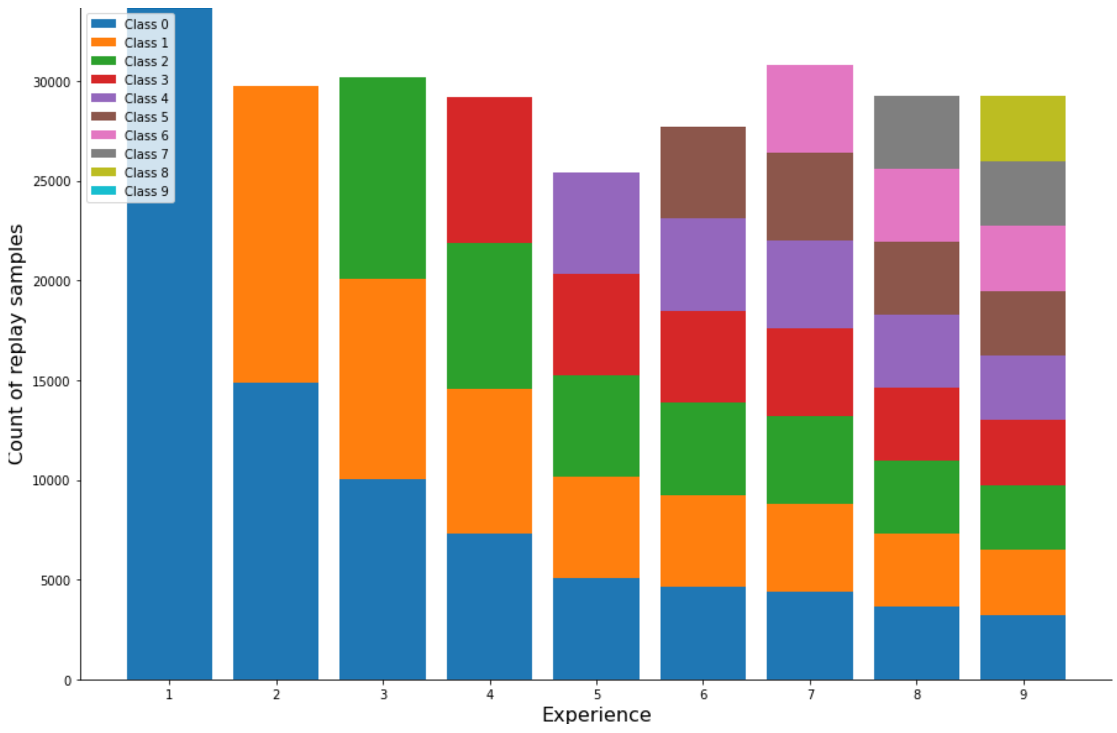

Figure 27: **Balanced Replay Sample Distribution:** Distribution of replay labels among the ten classes. Plotted in a single bar for each experience (starting from experience 1; there are no replay samples used in experience 0).

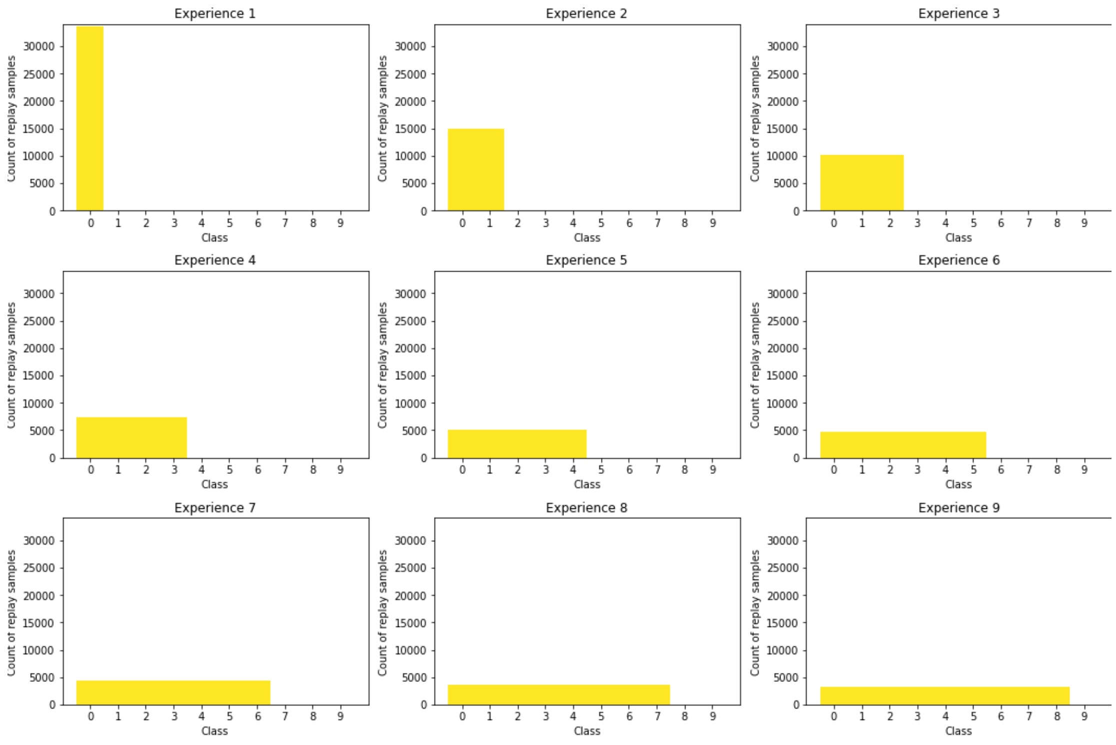

Figure 28: **Balanced Replay Sample Distribution:** Distribution of replay labels among the ten classes. Plotted in a separate bar plot for each experience (starting from experience 1; there are no replay samples used in experience 0).

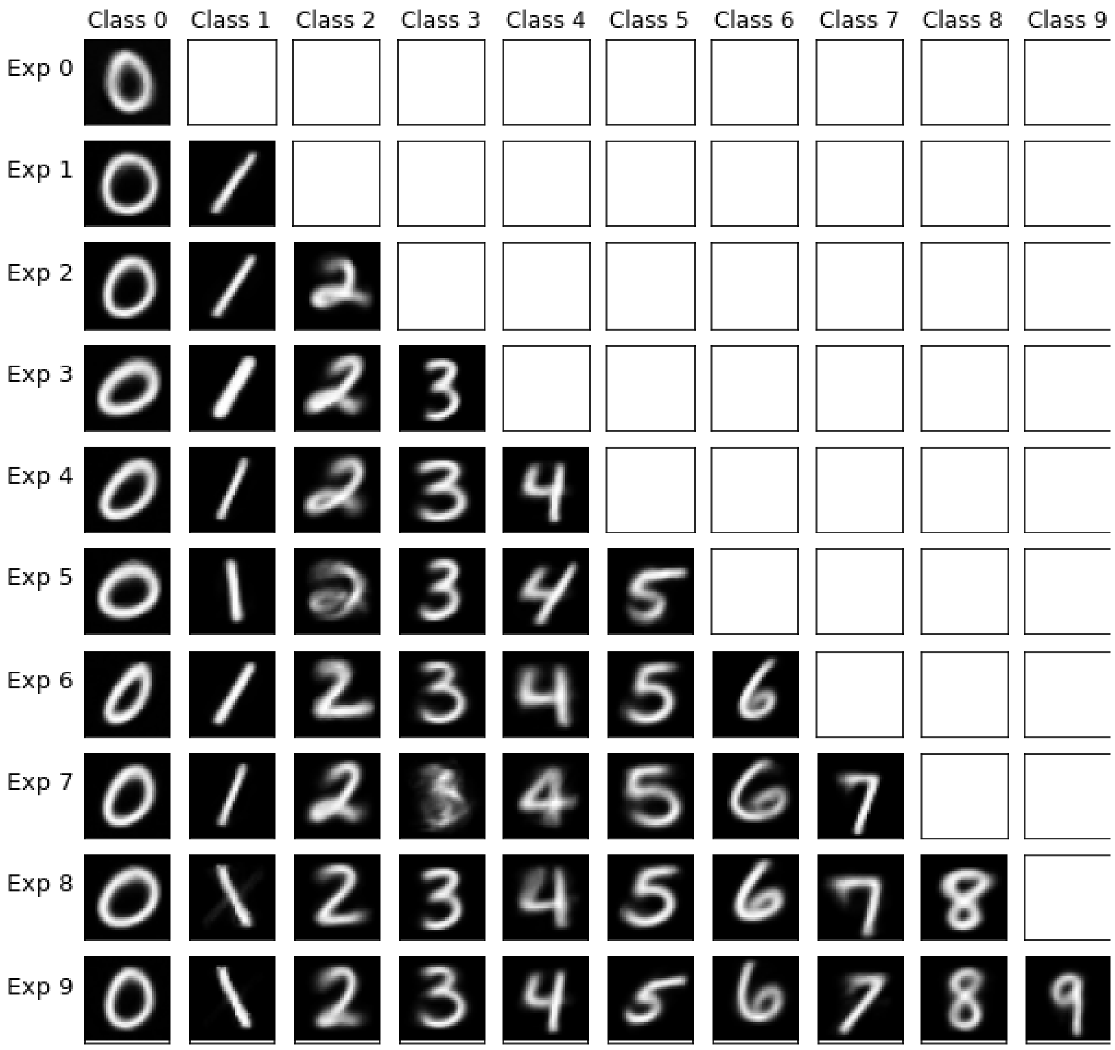

Figure 29: **Balanced Replay Sample Distribution:** Samples of images generated for each class after training an experience. White tiles mean that this class has not been encountered yet, grey tiles indicate that a class has been forgotten.