DDGR: Continual Learning with Deep Diffusion-based Generative Replay

Rui Gao 1  Weiwei Liu 1

Abstract

Popular deep-learning models in the field of image classification suffer from catastrophic forgetting—models will forget previously acquired skills when learning new ones. Generative replay (GR), which typically consists of a generator and a classifier, is an efficient way to mitigate catastrophic forgetting. However, conventional GR methods only focus on a single instruction relationship (generator-to-classifier), where the generator synthesizes samples for previous tasks to instruct the training of the classifier, while ignoring the ways in which the classifier can benefit the generator. In addition, most generative replay methods typically reuse the generated samples to update the generator, which causes the samples regenerated by the generator deviating from the distribution of previous tasks. To overcome these two issues, we propose a novel approach, called deep diffusion-based generative replay (DDGR), which adopts a diffusion model as the generator and calculates an instruction-operator through the classifier to instruct the generation of samples. Extensive experiments in class incremental (CI) and class incremental with repetition (CIR) settings demonstrate the advantages of DDGR. Our code is available at https://github.com/xiaocangshengGR/DDGR.

1. Introduction

In the digital world, the number of new tasks is increasing overwhelmingly as huge amounts of data are produced (Liu & Tsang, 2015; Liu et al., 2017; Liu & Tsang, 2017; Liu et al., 2019). This continuous generation of emerging learning tasks requires learning systems to adapt quickly to changing environments. Recently, many studies have shown that existing standard deep learning methods quickly forget their previously acquired experiences when learning new tasks (Kirkpatrick et al., 2017). This phenomenon, referred to as catastrophic forgetting (Kumaran et al., 2016), represents a significant challenge for scenarios in which tasks are learned in sequence and previous training data cannot be obtained.

Continual learning (CL)—also referred to as lifelong learning (Chen & Liu, 2018), sequential learning (Aljundi et al., 2019), and incremental learning (Aljundi et al., 2018)—has emerged as an efficient way to mitigate catastrophic forgetting in deep learning models. Classical CL works have explored a range of approaches to help the model remember previous knowledge. Among them, the generative replay (GR) model is one of the most popular methods due to its promising experimental results (Hayes et al., 2021). In the field of image classification, GR methods usually consist of two components: a generator and a classifier. The generator aims to generate or synthesize samples containing previous task knowledge and feeds these samples to the classifier. The classifier then trains with new data as well as generated samples to mitigate catastrophic forgetting. For example, Kemker & Kanan (2018) propose FearNet, which is a generator that employs a brain-inspired dual-memory system to instruct the classifier training; Wu et al. (2018) propose MeRGANs, which can generate images without forgetting; Ye & Bors (2020) propose L-VAEGAN, which induces generative replay samples and learns informative latent representations.

However, most GR methods suffer from two disadvantages: the adoption of instruction in a single direction (generator-to-classifier) and the reuse of generated samples. In CL, the classifier, which is trained on new data and previous samples, is able to help the generator adjust to the changing tasks. In contrast, most GR methods ignore the benefit offered by the classifier to the generator. Moreover, reusing generated samples may produce low-quality samples for previous tasks. For example, DGR (Shin et al., 2017) trains a new generator on a mixed data distribution of new samples and reused samples. Figure 1 shows that DGR generates fuzzy samples for previous tasks. This paper proposes to use a diffusion probabilistic model (referred to here for brevity as a diffusion model) (Sohl-Dickstein et al., 2015; Ho et al., 2020) to address these two issues.
Figure 1. We use DGR as an example to demonstrate that reusing generated samples will lead to the generation of low-quality samples for previous tasks. We conduct experiments on CORe50. Further details of these experiments are provided in Section 3.2.

Diffusion models are motivated by non-equilibrium thermodynamics (Sohl-Dickstein et al., 2015), and are proven to be able to synthesize high-quality images or audio (Ho et al., 2020; Chen et al., 2021; Ho et al., 2022a; Kong et al., 2021). As illustrated in Figure 2, a diffusion model maintains a Markov chain of diffusion steps to gradually add Gaussian noise to data, and learns to reverse the diffusion process in order to generate the desired samples from the noise. Different from VAE (Kingma & Welling, 2014) or GAN (Goodfellow et al., 2014), diffusion models are learned according to a fixed procedure and constructed by latent high-dimensional variables, which are scaled to be the same size as the original data.

In this paper, we propose a novel approach, called deep diffusion-based generative replay (DDGR), which uses a diffusion model as the generator to synthesize samples for previous tasks. DDGR concentrates on a bi-directional instructional relationship between the generator and the classifier. When learning new tasks, DDGR uses the classifier, which is pretrained on previous tasks, to instruct the sampling process of the diffusion model at each time step. New data is combined with the generated samples and used to train the classifier. The diffusion model is also trained on the combined data to enable it to adjust to the changing tasks. Meanwhile, the instruction provided by the classifier to the diffusion model guarantees a high quality of synthetic samples that will not decrease as the tasks change. Our main contributions can be summarized as follows:

- We focus on sampling process of a diffusion model and explore how this process might be instructed by a pretrained classifier. Specifically, we calculate instruction-operator through classifier at each time step of diffusion model to guide the generation of samples.
- The novel DDGR is proposed based on a diffusion model, where the classifier uses the instruction-operator to instruct the sampling process of the diffusion model. Benefiting from the instruction-operator, DDGR significantly improves the quality of generated samples for previous tasks.
- Extensive experimental results under class incremental (CI) and class incremental with repetition (CIR) settings demonstrate the advantages of DDGR.

2. Related Work

Continual learning. Existing researches in CL mainly focus on how to solve the catastrophic forgetting. Some surveys (Parisi et al., 2019; Lesort et al., 2019; Pfülb & Gepperth, 2019; Farquhar & Gal, 2018) have shown that existing CL works can be broadly divided into three categories: (i) regularization-based methods, (ii) parameter isolation methods and (ii) replay methods.

Classical regularization-based methods add an extra regularization term to the loss function. These methods penalize changes to important parameters of model for previous tasks (Kirkpatrick et al., 2017; Zenke et al., 2017; Aljundi et al., 2018; Lee et al., 2017). There are also other regularization-based methods that are based on functional regularization (Pan et al., 2020; Benjamin et al., 2019), uncertainty regularization (Ahn et al., 2019), classifier projection regularization (Cha et al., 2021) and node importance (Jung et al., 2020).

Parameter isolation methods dedicate different model parameters to each task to prevent any possible forgetting.
When no architectural size constraints apply, it is possible to grow new branches for new tasks while freezing previous task parameters (Xu & Zhu, 2018; Mallya & Lazebnik, 2018; Serrà et al., 2018).

Replay methods generally have two ways to replay in artificial neural networks: partial replay and generative replay (GR) (Hayes et al., 2021). Partial replay stores either all or a subset of previously learned inputs in a replay buffer, and mixes these inputs with new samples to train the classifier (Rebuffi et al., 2017; Lopez-Paz & Ranzato, 2017; Rolnick et al., 2019; Ayub & Wagner, 2021). In contrast to storing previous examples, GR generates synthetic samples for previous tasks. Classical GR methods use VAE or GAN as the generator for CL (Wu et al., 2018; Achille et al., 2018; Zhai et al., 2019; Wu et al., 2019; Ramapuram et al., 2020; Mundt et al., 2022). For example, DGR (Shin et al., 2017) uses GAN to generate previous samples for data replaying. Wu et al. (2018) propose MeRGANs to solve the problem of forgetting in generative models (GAN in particular), and achieve great performance in CL. In addition, Liu et al. (2020) propose a mnemonics training framework to generate exemplars which are optimizable. Most of these methods perform a single instruction and reuse the generated samples, which we try to solve in this paper. In addition, graph-based replay (Tang & Matteson, 2021) and gradient projection memory (Saha et al., 2021) have also been proven to be efficient.

**Diffusion model.** A diffusion model is a parameterized Markov chain trained using variational inference to produce samples matching the data after finite time. Diffusion models are based on two components: (i) a forward diffusion stage, in which the input data is gradually perturbed over several steps by adding Gaussian noise; (ii) a reverse diffusion stage, in which a generative model learns to gradually reverse the diffusion process to recover the input data from the diffused (noisy) data. Croitoru et al. (2022) provide a categorization of diffusion models based on the denoising condition: unconditional image generation and conditional image generation. Unconditional image generation do not require supervision signals, being completely unsupervised. For example, denoising diffusion probabilistic model (DDPM) (Ho et al., 2020), which learns the reverse process by estimating the noise in the image at each step, is a typical unconditional diffusion model: On top of DDPM, Nichol & Dhariwal (2021) propose a unconditional diffusion model which shows that it is required to learn the variance in terms of log-likelihood; the work of Nachmani et al. (2021) is also unconditional where they replace the Gaussian noise distributions with two other distributions, a mixture of two Gaussians and the Gamma distribution. Conditional image generation is commonly based on various source signals, in most cases some class labels being used. For example, Pandey et al. (2021) build a generator-refiner framework, where the generator is a VAE and the refiner is a DDPM conditioned by the output of the VAE; Ho & Salimans (2022) introduce a guidance method whose idea is based on the implicit classifier derived from the Bayes rule; Singh et al. (2022) propose a novel method for conditional image generation which conditions the noise signal (from where the sampling starts) instead of conditioning the signal throughout the sampling process. More details can be found in the work of Croitoru et al. (2022). Recently, diffusion models have shown their exquisite potential in the field of computer vision (Dhariwal & Nichol, 2021; Ho et al., 2022b; Li et al., 2022; Luo & Hu, 2021), audio processing (Chen et al., 2021; Popov et al., 2021; Kim et al., 2022), and AI for science (Luo et al., 2021; Xu et al., 2022; Jing et al., 2022). In this paper, we show the advantages of diffusion models in CL.

### 3. Preliminaries

#### 3.1. Denoising diffusion probabilistic model

Our work is based on DDPM (Ho et al., 2020) which is a typical diffusion model architecture. As shown in Figure 2, DDPM consists of a forward and a reverse processes.
Forward process. Given a data point sampled from a real data distribution \( x_0 \sim q(x_0) \), the forward process is fixed to a Markov chain, which gradually adds Gaussian noise to the data according to a variance schedule \( \{ \beta_n \in (0, 1) \}_{n=1}^{N} \):

\[
q(x_1:N \mid x_0) = \prod_{n=1}^{N} q(x_n \mid x_{n-1})
\] (1)

\[
q(x_n \mid x_{n-1}) = \mathcal{N}(\frac{x_n}{\sqrt{1-\beta_n}x_{n-1}}, \beta_n I)
\] (2)

where \( \sqrt{1-\beta_n}x_{n-1} \) and \( \beta_n I \) are the mean and variance of the Gaussian distribution respectively, while \( x_n \) is the noisy sample at time step \( n \). It is noteworthy that the mean, variance, and noisy samples have the same dimensionality as \( x_0 \). This process produces a sequence of noisy samples \( x_1, \ldots, x_N \) in \( N \) steps. Based on reparameterization trick, \( x_n \) at any time step \( n \) can be derived by \( x_0 \) as follows:

\[
x_n = \sqrt{\alpha_n} x_0 + \sqrt{1-\alpha_n} \epsilon
\] (3)

\[
q(x_n \mid x_0) = \mathcal{N}(\sqrt{\alpha_n} x_0, (1-\alpha_n) I)
\] (4)

where \( \epsilon \sim \mathcal{N}(0, I) \), \( \alpha_n = 1-\beta_n \) and \( \alpha_n = \prod_{i=1}^{n} \alpha_i \) (see Appendix B for details of Equations (3) and (4)).

Reverse process. If we can reverse the above process and generate samples from \( q(x_{n-1} \mid x_n) \), it is possible to construct the original input from the Gaussian noise. However, the conditional probability \( q(x_{n-1} \mid x_n) \) is unknown. An alternative is to train a model with parameter \( \theta \) to learn a conditional probability \( p_\theta(x_{n-1} \mid x_n) \) that approximates \( q(x_{n-1} \mid x_n) \). The reverse process can be expressed as follows:

\[
p_\theta(x_0:N) = p(x_N) \prod_{n=1}^{N} p_\theta(x_{n-1} \mid x_n)
\] (5)

\[
p_\theta(x_{n-1} \mid x_n) = \mathcal{N}(\mu_n, \sigma_n^2)
\] (6)

Training objective. The setup above is similar to VAE. To formulate this minimization problem, a variational lower bound is used to optimize the negative log-likelihood:

\[
-\log p_\theta(x_0) \leq -\log p_\theta(x_0) + D_{KL}(q(x_{1:N} \mid x_0) \| p_\theta(x_{1:N} \mid x_0))
\]

\[
= \mathbb{E}_{q(x_{1:N} \mid x_0)} \left[ \log \frac{q(x_{1:N} \mid x_0)}{p_\theta(x_{1:N} \mid x_0)} \right]
\] (7)

\[
-\mathbb{E}_{q(x_0)} \log p_\theta(x_0) \leq \mathbb{E}_{q(x_0)} \left[ \log \frac{q(x_{1:N} \mid x_0)}{p_\theta(x_{1:N} \mid x_0)} \right] =: L_{LB}
\] (8)

Sohl-Dickstein et al. (2015) split the objective \( L_{LB} \) into several KL-divergence terms, where each term in the objective is analytically computable:

\[
L_{LB} = \mathbb{E}_{q(x_{1:N} \mid x_0)} \left[ \log \frac{q(x_{1:N} \mid x_0)}{p_\theta(x_{1:N} \mid x_0)} \right]
\]

\[
= \mathbb{E}_{q} \left[ D_{KL}(q(x_{1:N} \mid x_0) \| p_\theta(x_{1:N} \mid x_0)) \right]
\]

\[
+ \sum_{n \geq 1} D_{KL}(q(x_n \mid x_{n+1}, x_0) \| p_\theta(x_n \mid x_{n+1}))
\]

\[
- \log p_\theta(x_1 \mid x_0)
\]

DDPM fixes \( \beta_n \) to constants, and \( L_N \) can be ignored because it is also constant. \( L_0 \) is modelled by a separate discrete decoder derived from \( \mathcal{N}(x_0; \mu_0, \sigma_0) \). As for \( L_n \), it is noteworthy that \( q(x_n \mid x_{n+1}, x_0) \) of \( L_n \) is traceable:

\[
q(x_n \mid x_{n+1}, x_0) = \mathcal{N}(x_n; \mu_{n+1}, \Sigma_{n+1})
\] (10)

\[
\mu_{n+1} = \frac{1}{\sqrt{1-\alpha_n+\beta_n}}(x_{n+1} - \frac{1}{\sqrt{1-\alpha_n}} \epsilon)
\] (11)

\[
\Sigma_{n+1} = \frac{1 - \alpha_n}{1 - \alpha_{n+1}} \beta_n I
\] (12)

In DDPM, \( \Sigma_{n+1} \) are set to \( \sigma^2_{n+1} I \), which are time dependent constants. As for \( \mu_{n+1} \), it is only necessary to train \( \epsilon_\theta(x_{n+1}, n+1) \) to predict \( \epsilon \), because \( x_{n+1} \) can be derived at training time. Therefore, the objective is:

\[
L_n = \mathbb{E}_{x_0, \epsilon} \left[ \omega \| \epsilon - \epsilon_\theta(x_{n+1}, n+1) \| ^2 \right] + C
\] (13)

where \( \omega = \frac{\sqrt{(1-\alpha_{n+1})^2}}{2}\sigma^2_{n+1} \) is the weighting term, while \( C \) is a constant term that does not depend on \( \theta \). \( \epsilon_\theta(x_{n+1}, n+1) \) is a model with parameter \( \theta \), which takes \( x_{n+1} \) and \( n+1 \) as inputs. Ho et al. (2020) find that it is better to train the diffusion model without the weighting term. The final objective can be expressed as follows:

\[
\mathbb{E}_{x_{1:N-1}, x_0, \epsilon} \left[ \| \epsilon - \epsilon_\theta(x_{n+1}, n+1) \| ^2 \right]
\] (14)

where \( x_{n+1} \) in Equation (13) is replaced according to Equation (3). Further details regarding the training objective can be found in Appendix C.

Sampling. After training the model \( \epsilon_\theta(x_{n+1}, n+1) \), the conditional probability, the mean and variance of which are:

\[
\frac{1}{\sqrt{\alpha_{n+1}}} (x_{n+1} - \frac{1-\alpha_{n+1}}{\sqrt{1-\alpha_{n+1}}} \epsilon_\theta(x_{n+1}, n+1)) \text{ and } \sigma^2_{n+1} I
\]

respectively, is known. Starting at \( x_N \sim \mathcal{N}(0, I) \), the complete sampling process can be run according to the following equation:

\[
x_n = \frac{1}{\sqrt{\alpha_{n+1}}} (x_{n+1} - \frac{1-\alpha_{n+1}}{\sqrt{1-\alpha_{n+1}}} \epsilon_\theta(x_{n+1}, n+1)) + \sigma_{n+1} z
\] (15)

where \( z \sim \mathcal{N}(0, I) \) and \( n = N - 1, \ldots, 0 \).
3.2. Preliminary experiments

To demonstrate that reusing generated samples will lead to low-quality generated samples for previous tasks, we conduct preliminary experiments on CORe50 (Lomonaco & Maltoni, 2017). Details of the setup on CORe50 can be found in Section 5.1. We take DGR as an example and run DGR twice on CORe50. In the first run, we assume that the generator has access to the previous task data, and use the original data of the previous tasks to update the generator. In the second run, we reuse the generated samples to update the DGR generator. We investigate the generated samples at each task over the two runs. Partial samples are presented in Figure 1. We find that the samples in the second run (in which the generated samples are reused) become blurred as new tasks are learned. In contrast, the samples in the first run can still be clearly distinguished as new tasks are learned. These results indicate that reusing the generated samples will decrease the quality of the generated samples when learning new tasks.

4. Deep diffusion-based generative replay

In this section, we present our proposed method—DDGR. We adopt DDPM as the generator. Moreover, we utilize a bi-direction relationship between the generator and the classifier. Similar to most GR methods, DDGR uses the generator to generate samples for use in training the classifier for previous tasks. In addition, DDGR also uses the classifier, which is pretrained on previous tasks, to instruct the generator to synthesize high-quality samples of previous tasks. Specifically, we calculate an instruction-operator, which affects the noisy sample, through the classifier at each time step of the diffusion model.

4.1. Instruction-operator

In classifier-to-generator instruction, we focus primarily on the sampling process of the diffusion model. Dhariwal & Nichol (2021) inspire us to achieve this by a classifier. We assume that the classifier \( f(x) \) is pretrained on previous sample set \( \mathcal{X} \) (drawn from \( D_{\text{pre}} \), which is the mixed distribution of previous tasks) with label set \( \mathcal{Y} = \{y_1, y_2, \ldots, y_m\} \). To mitigate catastrophic forgetting, the classifier requires high-quality generated samples whose labels belong to \( \mathcal{Y} \). Therefore, we need to consider how to condition the sample process of DDPM on a specific label \( y \).

We first condition the entire Markov process of DDPM on label \( y \), and define conditional Markov process as \( \hat{q} \) (to distinguish it from unconditional Markov process \( q \), which corresponds to DDPM). The key role of conditional sampling on \( y \) is to track the conditional probability \( \hat{q}(x_n | x_{n+1}, y) \). It is noteworthy that we have the following theorem:

**Theorem 4.1.** \( \hat{q}(x_n | x_{n+1}, y) \) is proportional to \( q(x_n | x_{n+1}) \hat{q}(y | x_n) \), where \( q(x_n | x_{n+1}) \) is the conditional probability in the reverse process of \( q \), and \( \hat{q}(y | x_n) \) is the label distribution given \( x_n \).

See Appendix D for more details. We assume that \( \hat{q}(y | x_n) \) in Theorem 4.1 is approximated by \( p_C(y | x_n) \) with parameter \( \zeta \). As introduced in Section 3.1, we use \( p_\theta(x_n | x_{n+1}) \) to approximate \( q(x_n | x_{n+1}) \) after training the diffusion model. By plugging \( p_C(y | x_n) \) and \( p_\theta(x_n | x_{n+1}) \) into Theorem 4.1, we can derive the following:

\[
\hat{q}(x_n | x_{n+1}, y) = C_{nor} p_\theta(x_n | x_{n+1}) p_C(y | x_n) \tag{16}
\]

where \( C_{nor} \) is a normalizing constant term. Recalling that the conditional probability in sampling process is \( p_\theta(x_n | x_{n+1}) = N(x_n; \mu_\theta(x_{n+1}, n + 1), \sigma_n^2 I) \), according to the Gaussian density function, we can then derive \( \log p_\theta(x_n | x_{n+1}) \) as follows:

\[
\log p_\theta(x_n | x_{n+1}) = C - \frac{(x_n - \mu_\theta)^2}{2 \sigma_n^2 I} \tag{17}
\]

where \( C \) is a constant term independent from \( x_n \), while \( \mu_\theta \) represents \( \mu_\theta(x_{n+1}, n + 1) \) for convenience. We also use \( C \) to represent all constant terms that are independent from \( x_n \) in the following steps for convenience. As for \( \log p_C(y | x_n) \), we use the Taylor expansion of \( \log p_C(y | x_n) \) at \( x_n = \mu_\theta \) as an approximation of itself.

\[
\log p_C(y | x_n) \approx \log p_C(y | x_n)_{|x_n=\mu_\theta} + (x_n - \mu_\theta) \nabla_{x_n} \log p_C(y | x_n)_{|x_n=\mu_\theta} = C + (x_n - \mu_\theta) G \tag{18}
\]

where \( G = \nabla_{x_n} \log p_C(y | x_n)_{|x_n=\mu_\theta} \). We can derive \( \log \hat{q}(x_n | x_{n+1}, y) \) by Equations (16), (17) and (18):

\[
\log \hat{q}(x_n | x_{n+1}, y) = \log \left( p_\theta(x_n | x_{n+1}) p_C(y | x_n) \right) + C \tag{19}
\]

\[
\approx \log \left( p_\theta(x_n | x_{n+1}) p_\theta(g(x_n)) \right) + C \tag{20}
\]

\[
\approx -\frac{(x_n - \mu_\theta)^2}{2 \sigma_n^2 I} + (x_n - \mu_\theta) G + C \tag{21}
\]

\[
= -(x_n - \mu_\theta - \sigma_n^2 G)^2 \frac{1}{2 \sigma_n^2 I} + \frac{\sigma_n^2}{2} G^2 + C \tag{22}
\]

\[
= -(x_n - \mu_\theta - \sigma_n^2 G)^2 \frac{1}{2 \sigma_n^2 I} + C \tag{23}
\]

\[
= -C - (x_n - \mu_\theta - \sigma_n^2 G) (2 \sigma_n^2 I)^{-1} (x_n - \mu_\theta - \sigma_n^2 G) \tag{24}
\]

where the constant term \( C \) can be ignored due to the normalizing constant \( C_{nor} \) in Equation (16). Recalling the Gaussian density distribution, Equation (24) means that \( \hat{q}(x_n | x_{n+1}, y) \) can be approximated by...
where \( \mu \) where generate the desirable samples for the classifiers. sample at each time step, enabling the diffusion model to calculate the diffusion model generates samples from Gaussian noise, we of the diffusion model can be found in Figure 3. When the instruction-operator works in the sample process of the diffusion model. At each time step, we perform steps (1) and (2) to calculate an instruction-operator through the classifier. We plug the instruction-operator into the sample transition at step (3) and derive the next noisy sample \( x_n \).

\[
\mathcal{N} \left( x_n; \mu_0 + \sigma_{n+1}^2 G, \sigma_{n+1}^2 I \right). \quad \text{The condition of the sampling process given label } y \text{ as presented below:}
\]

\[
x_n = \mu_0 + \sigma_{n+1}^2 G + \sigma_{n+1}^2 z
\]  

(25)

where \( \mu_0 = \frac{1}{\sqrt{\sigma_{n+1}}} \left( x_{n+1} - \frac{1 - \alpha_{n+1}}{1 - \alpha_n} \epsilon_\theta \left( x_{n+1}, n+1 \right) \right) \) and \( z \sim \mathcal{N}(0, I) \).

Let us now turn our attention to the operator \( G = \nabla_{x_n} \log p_\zeta \left( y \mid x_n \right) \big|_{x_n = \mu_0} \). In the operator, \( p_\zeta \left( y \mid x_n \right) \) is essentially a classifier that takes \( x_n \) as input and approximates \( \hat{q} \left( y \mid x_n \right) \) with parameter \( \zeta \). We assume that \( x_n \) is able to be drawn from \( D_{pr} \), which indicates that \( p_\zeta \left( y \mid x_n \right) \) and the pretrained classifier \( f \) have similar input domains. Therefore, we regard the pretrained classifier \( f \) as \( p_\zeta \left( y \mid x_n \right) \); this means that we can utilize the pretrained classifier \( f \) to instruct the sampling process of the diffusion model, as follows:

\[
G = \nabla_{x_n} \log p_\zeta \left( y \mid x_n \right) \big|_{x_n = \mu_0} = \nabla_{x_n} \ell \left( f_\zeta \left( x_n \right), y \right) \big|_{x_n = \mu_0}
\]  

(26)

where \( \ell (\cdot) \) is the loss function. We refer to this operator (26) as the instruction-operator, which plays an important role in classifier-to-generator instruction. An illustration of how the instruction-operator works in the sample process of the diffusion model can be found in Figure 3. When the diffusion model generates samples from Gaussian noise, we calculate the instruction-operator through the pretrained classifier \( f_\zeta \) and plug it into the sample transition (i.e. Equation (25)). This instruction-operator affects the noisy sample at each time step, enabling the diffusion model to generate the desirable samples for the classifiers.

4.2. Generative replay with instruction-operator

Overview. Here, we provide an overview of our proposed method DDGR in Algorithm 1. DDGR uses DDPM as the generator and adopts a bi-directional instruction relationship between the generator and the classifier. It is noteworthy that the previous classifier \( f_{\zeta_{t-1}} \) was trained in sequence on tasks 1, 2, \ldots, \( t-1 \), which indicates that \( f_{\zeta_{t-1}} \) can be regarded as the pretrained classifier \( f_\zeta \). When learning a new task, DDGR first uses a classifier pretrained on previous tasks to instruct the diffusion model to generate samples with previous labels (Line 4 of Algorithm 1). Specifically, we calculate an instruction-operator through the classifier at each time step of the diffusion model and plug it into the sample transition (details provided in Algorithm 2). After several iterations, the diffusion model generates the desirable samples with previous labels for the classifier. DDGR then trains the classifier with a combination of generated samples and new data (Line 9 of Algorithm 1). Finally, DDGR updates the diffusion model according to Equation (14) (Lines 10 to 17 of Algorithm 1).

Generated sample reuse issue. As shown in line 12 of Algorithm 1, DDGR also reuses the generated samples to update the diffusion model as the tasks change. However, benefiting from the instruction of the classifier, DDGR significantly improves the quality of the samples of previous tasks produced by the generator. When learning a new task, the classifier is a pretrained model, which means that the classifier contains rich knowledge of previous tasks. The instruction-operator can be intuitively regarded as a type of
We conduct extensive experiments to verify the superior performance of our proposed DDGR. In this paper, we consider two scenarios commonly encountered in CL, namely CI (van der Ven & Tolas, 2018; van de Ven & Tolas, 2019) and CIR (Lomonaco & Maltoni, 2017; Cosso et al., 2022).

5.1. Experimental setup

Baselines: In order to verify the superiority of our proposed method, we select eight baselines for comparison:

- SI (Zenke et al., 2017), MAS (Aljundi et al., 2018), EWC (Kirkpatrick et al., 2017), IMM (Lee et al., 2017), DGR (Shin et al., 2017), MeRGAN (Wu et al., 2018), PASS (Zhu et al., 2021) and Finetuning. SI, MAS, EWC, and IMM can be considered as the prior-focused methods, which estimate a distribution of model parameters as the prior when learning from new data. These methods typically estimate the importance of all neural network parameters, with parameters assumed to be independent to ensure feasibility. DGR and MeRGAN use GAN to generate previous samples for data replaying. PASS is a simple non-exemplar based method. See Appendix A for details.

Datasets: In CI scenario, we conduct experiments on two widely used datasets: CIFAR-100 and ImageNet (Deng et al., 2009). For this scenario, we follow Liu et al. (2020) and utilize a similar division of datasets, which has one initial task and four incremental tasks. For CIFAR-100, initial task consists of 50 random classes, and all incremental tasks have the same number of classes: specifically, the number of classes in one incremental task is set to either 5 or 10. ImageNet contains around 1.3 million samples of 224 × 224 color images from 1000 classes. In ImageNet, initial task consists of 500 random classes, and the number of classes in one incremental task is set to either 50 or 100. Moreover, in CIR scenario, we use CORE50 (Lomonaco & Maltoni, 2017) to conduct experiments. The data in CORE50 was collected during eleven distinct sessions (eight indoor and three outdoor) characterized by different backgrounds and lighting conditions. The dataset consists of 50 domestic objects belonging to 10 categories. Classification in this paper is performed using a 10-way 1-shot classification setting.
DDGR: Continual Learning with Deep Diffusion-based Generative Replay

Table 1. All results on two datasets in the CI scenario. We present the final average accuracy $A_T$ and forgetting rate $F_T^{avg}$.

<table>
<thead>
<tr>
<th>Method</th>
<th>CIFAR-100</th>
<th>ImageNet</th>
<th>CIFAR-100</th>
<th>ImageNet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AlexNet</td>
<td>ResNet</td>
<td>AlexNet</td>
<td>ResNet</td>
</tr>
<tr>
<td>Finetuning</td>
<td>6.11</td>
<td>5.12</td>
<td>18.08</td>
<td>17.50</td>
</tr>
<tr>
<td>SI</td>
<td>16.96</td>
<td>13.57</td>
<td>26.45</td>
<td>23.15</td>
</tr>
<tr>
<td>EWC</td>
<td>15.29</td>
<td>9.71</td>
<td>25.49</td>
<td>18.82</td>
</tr>
<tr>
<td>MAS</td>
<td>20.13</td>
<td>18.94</td>
<td>29.94</td>
<td>28.28</td>
</tr>
<tr>
<td>IMM</td>
<td>11.26</td>
<td>9.87</td>
<td>21.02</td>
<td>19.79</td>
</tr>
<tr>
<td>DGR</td>
<td>42.49</td>
<td>38.16</td>
<td>52.96</td>
<td>48.94</td>
</tr>
<tr>
<td>MeRGAN</td>
<td>46.03</td>
<td>43.23</td>
<td>57.19</td>
<td>55.69</td>
</tr>
<tr>
<td>PASS</td>
<td>53.21</td>
<td>48.65</td>
<td>62.30</td>
<td>60.64</td>
</tr>
<tr>
<td>DDGR</td>
<td>59.20</td>
<td>52.22</td>
<td>65.40</td>
<td>60.04</td>
</tr>
</tbody>
</table>

performed at object level (50 classes). We follow Lomonaco & Maltoni (2017), which generates 79 tasks from CORE50. The first task includes ten classes, and each subsequent task contains five classes. See Appendix A for details.

Models: For the classifier, we follow the work of Rebuffi et al. (2017); Liu et al. (2020); Delange et al. (2021) and use a 32-layer ResNet and AlexNet (Krizhevsky et al., 2012) for all experiments. Moreover, following the work of Ho et al. (2020), we use the UNet architecture for the diffusion model. The UNet model uses a stack of residual layers and downsampling convolutions, followed by a stack of residual layers with upsampling convolutions, with skip connections connecting layers with the same spatial size.

Evaluation: In the CI scenario, we use two evaluation metrics which are similar to the work of Liu et al. (2020); Choi et al. (2021). The average accuracy at task $i$ is defined as $A_i = acc_{C \cup i, i}$, where the learned model $M_i$ is evaluated on the test dataset $S_{test}^c \cup S_{test}^i$, containing all data (classes) seen so far. The forgetting rate of the previous class $c$ at task $i$ is defined as $F_{i,c} = acc_{j,c} - acc_{i,c}$, where $acc_{j,c}$ and $acc_{i,c}$ are the accuracies of models $M_j$ and $M_i$ on the test dataset $S_{test}^c$, respectively. $M_j$ is the first model to see class $c$, and $S_{test}^c$ is the dataset with only one label $c$. The average forgetting rate at task $i$ is defined as $F_{avg} = \sum_{c \in C_i} F_{i,c} / |C_i|$, where $C_i$ is the set of all seen classes. In the CI scenario, we follow the work of Lomonaco & Maltoni (2017) and evaluate the learned model $M_i$ by the average accuracy $A_{FTS} = acc_{i,FTS}$ which is the accuracy of $M_i$ on a Full Test Set (Lomonaco & Maltoni, 2017). The Full Test Set is fixed and includes the patterns of all classes.

5.2. Results in CI scenario

We gather the final average accuracies and average forgetting rates of all experiments in the CI scenario in Table 1; for convenience, we refer to them as accuracy and forgetting, respectively. Moreover, to maintain consistency with Liu et al. (2020), we refer to each task as a “phase” in the CI scenario. On CIFAR-100, DDGR achieves the best performance in almost all experiments. For example, when $NC$ is 5 and the model is AlexNet, we observe that DDGR reduces the forgetting from 24.08% to 23.00% compared with the best baseline, representing an improvement of 1.08%, while DDGR outperforms the best baseline by 5.99% in terms of accuracy. We also track the performance of each baseline on CIFAR-100 in Figures 4(a) to 4(d). As the figures clearly show, our proposed DDGR outperforms the other baselines. For example, when $NC$ is 5 and the model is AlexNet, the accuracy of DDGR is 0.05% to 10.04% higher than that of the best baseline at each phase; here, the accuracy of DDGR at each phase ranges from 58.57% to 66.13%.

Table 1 further presents the results of all experiments on ImageNet. As is evident, DDGR performs better than all seven baselines. For example, when $NC$ is 50 and the model is AlexNet, we find that DDGR improves the forgetting from 17.31% to 6.98% compared with the best baseline, representing an improvement of 10.33%. In the same experiment, DDGR outperforms the best baseline by 9.92% in terms of accuracy, achieving an overall accuracy of 53.86%. Figures 4(e) to 4(h) present the performance of all baselines on ImageNet. DDGR outperforms all baselines in every phase in all experiments. For example, when $NC$ is 50 and the model is AlexNet, the accuracy of DDGR is 0.20% to 9.92% higher than that of the best baseline at each phase; here, the accuracy of DDGR at each phase ranges from 53.68% to 62.49%.

5.3. Results in CIR scenario

To maintain consistency with related works (Lomonaco & Maltoni, 2017), we refer to each task as a “batch” in this scenario. As illustrated in Figure 5, the performance of the model trained by each baseline is not stable, but the accuracy $acc_{i,FTS}$ tends to increase with new batches. This means that the model can accumulate the knowledge of
DDGR: Continual Learning with Deep Diffusion-based Generative Replay

Figure 4. The curve of each baseline with different classifiers on CIFAR-100 and ImageNet. On CIFAR-100, there are 50 classes in the 0–th phase, and the number of classes (NC) of each subsequent phase is set to either 5 or 10. On ImageNet, there are 500 classes in the 0–th phase, and the number of classes (NC) of each subsequent phase is set to either 50 or 100.

Figure 5. The results of using AlexNet and ResNet on CORe50 in CIR scenario. There are ten classes in the 0–th batch, and five classes in each subsequent phase.

In addition, we demonstrate the advantages of DDGR in the experiments without a large first task (see more details in Appendix F). We present the effect of diffusion steps for DDGR in Appendix G. We also show that DDGR should be improved in terms of time cost in Appendix H.

6. Conclusion

In this paper, we propose a novel GR method called deep diffusion-based generative replay (DDGR). DDGR uses a diffusion model as the generator and proposes the instruction-operator to instruct the diffusion model, thereby generating high-quality samples for previous tasks. We demonstrate the advantages of DDGR by means of extensive experiments on CIFAR-100, ImageNet, and CORe50.

Acknowledgements

This work is supported by the National Natural Science Foundation of China under Grant 61976161, the Fundamental Research Funds for the Central Universities under Grant 2042022rc0016.
References


A. Additional experimental setting

Here, we provide more information about baselines and datasets. There are 7 baselines in our paper. We give some descriptions of these baselines and introduce the datasets.

SI: The synaptic state tracks the past and current parameter value, and maintains an online estimate of the synapse’s importance toward solving problems encountered in the past.

MAS: Redefines the parameter importance measure to an unsupervised setting and obtains gradients of the squared \( L_2 \)-norm of the learned network output function.

EWC: EWC introduces network parameter uncertainty in the Bayesian framework; the true posterior is estimated using a Laplace approximation with precision, determined by the Fisher Information Matrix (FIM), which shows equivalence to the positive semi-definite second order derivative of the loss near a minimum.

IMM: IMM estimates Gaussian posteriors for task parameters, in the same vein as EWC, but inherently differs in its use of model merging. In the merging step, the mixture of Gaussian posteriors is approximated by a single Gaussian distribution and corresponding covariances.

DGR: DGR consists of a deep generative model (“generator”) and a task solving model (“solver”). DGR trains a deep generative model in the generative adversarial networks (GANs) framework to mimic previous data. Generated data are then paired with corresponding response from the previous task solver to represent old tasks. Called the scholar model, the generator-solver pair can produce fake data and desired target pairs as much as needed, and when presented with a new task, these produced pairs are interleaved with new data to update the generator and solver networks.

MeRGAN: MeRGAN is a conditional GAN framework that integrates a memory replay generator. It contains two methods to prevent forgetting by leveraging replays, namely joint training with replay and replay alignment. In these paper, we use replay alignment.

PASS: PASS mainly consists of Prototype Augmentation and Self-Supervision. On the one hand, prototype augmentation (protoAug) memorizes one class representative prototype (typically the class mean in the deep feature space) for each old class, and augments the memorized prototypes via Gaussian noise when learning new classes. Then, the augmented prototypes and deep features of new data are jointly classified to maintain the discrimination and balance between old and new classes.

Finetuning: Finetuning greedily trains each task without considering previous task performance—hence introducing catastrophic forgetting—and represents the minimum desired performance.

<table>
<thead>
<tr>
<th>Table A.1. Details of datasets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks</td>
</tr>
<tr>
<td>CIFAR-100</td>
</tr>
<tr>
<td>ImageNet</td>
</tr>
<tr>
<td>CORe50</td>
</tr>
</tbody>
</table>

Datasets: There are three datasets in our work, we divide them into several tasks which can be found in Table A.1. In CI, we divide the datasets into 5 tasks where there are 1 initial task and 4 incremental tasks. For CIFAR-100, the initial task consist of 50 random classes, and all incremental tasks have same numbers of classes. The number of classes in one incremental task is set to be 5 or 10. For ImageNet, the initial task consist of 500 random classes, and the number of classes in one incremental task is set to be 50 or 100. In CIR, CORRe50 is a collection of 50 domestic objects belonging to 10 categories. Classification in this paper is performed at object level (50 classes). We follow up the work of (Lomonaco & Maltoni, 2017) which generates 79 tasks from CORRe50. The first task includes 10 classes, and each subsequent task contains 5 classes.
B. Reparameterization trick

In this section, we provide details of reparameterization trick in forward process of DDPM. A nice property of the forward process is that we can sample \( x_n \) at any time step \( n \) in a closed form using reparameterization trick:

\[
x_n = \sqrt{\alpha_n} x_{n-1} + \sqrt{1 - \alpha_n} \epsilon_{n-1}
\]  \( (B.1) \)

\[
x_{n-1} = \sqrt{\alpha_{n-1}} x_{n-2} + \sqrt{1 - \alpha_{n-1}} \epsilon_{n-2}
\]  \( (B.2) \)

where \( \alpha_n = 1 - \beta_n \) and \( \epsilon_{n-1}, \epsilon_{n-2} \sim \mathcal{N}(0, I) \). Plugging \((B.2)\) into \((B.1)\), we derive that:

\[
x_n = \sqrt{\alpha_n \alpha_{n-1}} x_{n-2} + \sqrt{\alpha_n (1 - \alpha_{n-1})} \epsilon_{n-2} + \sqrt{1 - \alpha_n} \epsilon_{n-1}
\]  \( (B.3) \)

where \( \sqrt{\alpha_n (1 - \alpha_{n-1})} \epsilon_{n-2} \) corresponds to \( \mathcal{N}(0, \alpha_n (1 - \alpha_{n-1}) I) \) and \( \sqrt{1 - \alpha_n} \epsilon_{n-1} \) corresponds to \( \mathcal{N}(0, 1 - \alpha_n I) \).

By merging these two distributions, we have the new distribution \( \mathcal{N}(0, 1 - \alpha_n I) \) with merged standard deviation \( \sqrt{\alpha_n (1 - \alpha_{n-1}) + (1 - \alpha_n)} = \sqrt{1 - \alpha_n \alpha_{n-1}} \). \((B.3)\) can be rewritten as:

\[
x_n = \sqrt{\alpha_n \alpha_{n-1}} x_{n-2} + \sqrt{1 - \alpha_n \alpha_{n-1}} \epsilon_{n-2}
\]  \( (B.4) \)

where \( \sqrt{1 - \alpha_n \alpha_{n-1}} \epsilon_{n-2} \) merges two Gaussians \( \mathcal{N}(0, \alpha_n (1 - \alpha_{n-1}) I) \) and \( \mathcal{N}(0, 1 - \alpha_n I) \). After \( n \) iterations, we can use \( x_0 \) to represent \( x_n \):

\[
x_n = \sqrt{\alpha_n} x_0 + \sqrt{1 - \alpha_n} \epsilon
\]  \( (B.5) \)

where \( \alpha_n = \prod_{i=1}^n \alpha_i \). These also indicate that:

\[
q(x_n | x_0) = \mathcal{N}(x_n; \sqrt{\alpha_n} x_0, (1 - \alpha_n) I)
\]  \( (B.6) \)

C. Additional derivation of training objective

There are two ways to derive the lower bound \( L_{LB} \). First, as shown in the main file, we can leverage the variational lower bound to optimize the negative log-likelihood:

\[
- \log p_\theta (x_0) \leq - \log p_\theta (x_0) + D_{KL}(q(x_{1:N} | x_0) \parallel p_\theta (x_{1:N} | x_0))
\]  \( (C.7) \)

\[
= - \log p_\theta (x_0) + \mathbb{E}_{x_{1:N} \sim q(x_{1:N} | x_0)} \left[ \log \frac{q(x_{1:N} | x_0)}{p_\theta (x_{0:N} | x_0)} \right]
\]  \( (C.8) \)

\[
= - \log p_\theta (x_0) + \mathbb{E}_{q(x_{1:N} | x_0)} \left[ \log \frac{q(x_{1:N} | x_0)}{p_\theta (x_{0:N} | x_0)} + \log p_\theta (x_0) \right]
\]  \( (C.9) \)

\[
= \mathbb{E}_{q(x_{1:N} | x_0)} \left[ \log \frac{q(x_{1:N} | x_0)}{p_\theta (x_{0:N} | x_0)} \right]
\]  \( (C.10) \)

\[
- \mathbb{E}_{q(x_{0:N})} \log p_\theta (x_0) \leq \mathbb{E}_{q(x_{0:N})} \left[ \log \frac{q(x_{1:N} | x_0)}{p_\theta (x_{0:N} | x_0)} \right] =: L_{LB}
\]  \( (C.11) \)

The other way to derive \( L_{LB} \) is to minimize the cross-entropy as the training objective:

\[
L_{CE} = - \mathbb{E}_{q(x_{0:N})} \log p_\theta (x_0)
\]  \( (C.12) \)

\[
= - \mathbb{E}_{q(x_{0:N})} \log \left( \int p_\theta (x_{0:N}) dx_{1:N} \right)
\]  \( (C.13) \)

\[
= - \mathbb{E}_{q(x_{0:N})} \log \left( \int q(x_{1:N} | x_0) \frac{p_\theta (x_{0:N})}{q(x_{1:N} | x_0)} dx_{1:N} \right)
\]  \( (C.14) \)

\[
= - \mathbb{E}_{q(x_{0:N})} \log \left( \frac{p_\theta (x_{0,N})}{q(x_{1:N} | x_0)} \right)
\]  \( (C.15) \)

\[
\leq - \mathbb{E}_{q(x_{0:N})} \log \frac{p_\theta (x_{0:N})}{q(x_{1:N} | x_0)}
\]  \( (C.16) \)

\[
= \mathbb{E}_{q(x_{0:N})} \left[ \log \frac{q(x_{1:N} | x_0)}{p_\theta (x_{0:N})} \right] = L_{LB}
\]  \( (C.17) \)
where (C.16) uses the Jensen’s inequality directly. Sohl-Dickstein et al. (2015) split the objective $L_{LB}$ into several KL-divergence terms where each term in the objective is analytically computable:

$$L_{LB} = \mathbb{E}_{q(x_{n:N})} \left[ \log \frac{q(x_{1:N} \mid x_0)}{p_\theta(x_{0:N})} \right] = \mathbb{E}_{q} \left[ \log \frac{\prod_{n=0}^{N-1} q(x_{n+1} \mid x_n)}{p_\theta(x_N)} \frac{q(x_{n+1} \mid x_n)}{p_\theta(x_n \mid x_{n+1})} \right]$$

$$= \mathbb{E}_{q} \left[ - \log p_\theta(x_N) + \sum_{n=0}^{N-1} \log \frac{q(x_{n+1} \mid x_n)}{p_\theta(x_n \mid x_{n+1})} \right]$$

$$= \mathbb{E}_{q} \left[ - \log p_\theta(x_N) + \sum_{n=0}^{N-1} \log \frac{q(x_{n+1} \mid x_n)}{p_\theta(x_n \mid x_{n+1})} + \log \frac{q(x_1 \mid x_0)}{p_\theta(x_0 \mid x_1)} \right]$$

$$= \mathbb{E}_{q} \left[ - \log p_\theta(x_N) + \sum_{n=0}^{N-1} \log \left( \frac{q(x_n \mid x_{n+1}, x_0)}{p_\theta(x_n \mid x_{n+1})} \cdot \frac{q(x_{n+1} \mid x_0)}{q(x_n \mid x_0)} \right) + \log \frac{q(x_1 \mid x_0)}{p_\theta(x_0 \mid x_1)} \right]$$

$$= \mathbb{E}_{q} \left[ - \log p_\theta(x_N) + \sum_{n=0}^{N-1} \log \left( \frac{q(x_n \mid x_{n+1}, x_0)}{p_\theta(x_n \mid x_{n+1})} + \log \frac{q(x_n \mid x_0)}{q(x_n \mid x_0)} \right) + \log \frac{q(x_1 \mid x_0)}{p_\theta(x_0 \mid x_1)} \right]$$

$$= \mathbb{E}_{q} \left[ \log \frac{q(x_N \mid x_0)}{p_\theta(x_N)} + \sum_{n=0}^{N-1} \log \frac{q(x_n \mid x_{n+1}, x_0)}{p_\theta(x_n \mid x_{n+1})} - \log p_\theta(x_0 \mid x_1) \right]$$

$$= \mathbb{E}_{q} \left[ D_{KL}(q(x_N \mid x_0) \parallel p(x_N)) \sum_{n=0}^{N-1} D_{KL}(q(x_n \mid x_{n+1}, x_0) \parallel p_\theta(x_n \mid x_{n+1})) - \log p_\theta(x_0 \mid x_1) \right].$$

DDPM fixes $\beta_n$ to constants. $L_N$ can be ignored because $L_N$ is also constant—no learnable parameters. In the work of Ho et al. (2020), $L_0$ is modelled by a separate discrete decoder derived from $N(x_0; \mu_\theta(x_1), \Sigma_\theta(x_1))$. As for $L_n$ ($n = 1, \ldots, N-1$), it is worth noting that $q(x_n \mid x_{n+1}, x_0)$ of $L_n$ is traceable:

$$q(x_n \mid x_{n+1}, x_0) = N(x_n; \mu_{n+1}, \Sigma_{n+1})$$

According to Bayes’ rule, we can derive that:

$$q(x_n \mid x_{n+1}, x_0) = q(x_{n+1} \mid x_n, x_0) \cdot q(x_n \mid x_0) \frac{q(x_{n+1} \mid x_n)}{q(x_n \mid x_{n+1})}$$

$$= \exp \left( -\frac{1}{2} \left( \frac{(x_{n+1} - \sqrt{\alpha_{n+1}} x_n)^2}{\beta_{n+1}} + (x_n - \sqrt{\alpha_n} x_0)^2 \frac{1}{1 - \alpha_n} - \frac{(x_{n+1} - \sqrt{\alpha_{n+1}} x_0)^2}{1 - \alpha_{n+1}} \right) \right)$$

$$= \exp \left( -\frac{1}{2} \left( \frac{(x_{n+1}^2 - 2\sqrt{\alpha_{n+1}} x_n x_0 + \alpha_{n+1} x_0^2 \beta_{n+1}}{\beta_{n+1}} + \frac{x_n^2 - 2\sqrt{\alpha_n} x_n x_0 + \alpha_n x_0^2}{1 - \alpha_n} \frac{1}{1 - \alpha_{n+1}} \right) \right)$$

$$= \exp \left( -\frac{1}{2} \left( \frac{x_{n+1}^2}{\beta_{n+1}} + \frac{1}{1 - \alpha_n} \frac{1}{\beta_{n+1}} \frac{x_n^2}{2\alpha_{n+1} x_0 + \sqrt{\alpha_{n+1}} x_0} + C(x_{n+1}, x_0) \right) \right)$$
where $C(x_{n+1}, x_0)$ is not involved with $x_n$ and can be ignored. According to the standard Gaussian density function, we can achieve the mean:

$$\bar{\mu}_{n+1} = \frac{\sqrt{\alpha_{n+1}}}{\beta_{n+1}} x_{n+1} + \frac{\sqrt{\alpha_n}}{1 - \alpha_n} x_0 \left/ \left( \frac{\alpha_{n+1}}{\beta_{n+1}} + \frac{1}{1 - \alpha_n} \right) \right.$$

$$= \left( \frac{\sqrt{\alpha_{n+1}}}{\beta_{n+1}} x_{n+1} + \frac{\sqrt{\alpha_n}}{1 - \alpha_n} x_0 \right) \frac{1}{\beta_{n+1}} \left/ \frac{1}{1 - \alpha_n} \cdot \beta_{n+1} \right.$$

$$= \frac{\sqrt{\alpha_{n+1}}}{1 - \alpha_n} x_{n+1} + \frac{\sqrt{\alpha_n \beta_{n+1}}}{1 - \alpha_n} x_0 \left/ \beta_{n+1} \right.$$  \hspace{1cm} (C.33)

Recall that $x_{n+1} = \sqrt{\alpha_{n+1}} x_0 + \sqrt{1 - \alpha_{n+1}} \epsilon$, we have $x_0 = \frac{1}{\sqrt{\alpha_{n+1}}} (x_{n+1} - \sqrt{1 - \alpha_{n+1}} \epsilon)$. Plugging it in (C.35), we derive that:

$$\bar{\mu}_{n+1} = \frac{\sqrt{\alpha_{n+1}}}{1 - \alpha_n} x_{n+1} + \frac{\sqrt{\alpha_n \beta_{n+1}}}{\alpha_{n+1}} \frac{1}{\sqrt{1 - \alpha_{n+1}}} (x_{n+1} - \sqrt{1 - \alpha_{n+1}} \epsilon)$$

$$= \frac{1}{\sqrt{\alpha_{n+1}}} x_{n+1} - \frac{1 - \alpha_{n+1}}{\sqrt{1 - \alpha_{n+1}}} \epsilon$$  \hspace{1cm} (C.36)

The variance can also be achieved:

$$\tilde{\Sigma}_{n+1} = 1/ \left( \frac{\alpha_{n+1}}{\beta_{n+1}} + \frac{1}{1 - \alpha_n} \right) = \frac{1}{1 - \alpha_n} \cdot \beta_{n+1}$$

$$= \frac{1 - \alpha_n}{1 - \alpha_n} \cdot \beta_{n+1}$$  \hspace{1cm} (C.37)

In DDPM, $\tilde{\Sigma}_{n+1}$ is set to $\sigma_n^2 \mathbb{I}$ which are untrained time dependent constants. These mean that it only needs to train $\mu_{\theta}(x_{n+1}, n + 1)$ to predict $\tilde{\mu}_{n+1}$. As $x_{n+1}$ can be achieved at training time, it only needs to train $\epsilon_{\theta}(x_{n+1}, n + 1)$ to predict $\epsilon$. By using a specific parameterization, we achieve:

$$L_n = \mathbb{E}_q \left[ \frac{1}{2\tilde{\Sigma}_{n+1}} \right] \left[ \bar{\mu}_{n+1} - \mu_{\theta}(x_{n+1}, n + 1) \right] \left/ \left( \frac{\alpha_{n+1}}{\beta_{n+1}} + \frac{1}{1 - \alpha_n} \right) \right.$$  \hspace{1cm} (C.38)

$$= \mathbb{E}_{x_0, \epsilon} \left[ \frac{1}{2\tilde{\Sigma}_{n+1}} \right] \left[ \left( \frac{1}{\sqrt{\alpha_{n+1}}} x_{n+1} - \frac{1 - \alpha_{n+1}}{\sqrt{1 - \alpha_{n+1}}} \epsilon \right) - \frac{1}{\sqrt{1 - \alpha_{n+1}}} \epsilon \right] \left/ \frac{1 - \alpha_{n+1}}{\sqrt{1 - \alpha_{n+1}}} \epsilon \right.$$  \hspace{1cm} (C.39)

$$= \mathbb{E}_{x_0, \epsilon} \left[ \frac{1}{2\tilde{\Sigma}_{n+1} \alpha_{n+1} (1 - \alpha_{n+1})} \right] \left[ \left( 1 - \alpha_{n+1} \right)^2 \epsilon - \epsilon_{\theta} \left( \sqrt{\alpha_{n+1}} x_0 + \sqrt{1 - \alpha_{n+1}} \epsilon, n + 1 \right) \right] \left/ \frac{1 - \alpha_{n+1}}{\sqrt{1 - \alpha_{n+1}}} \epsilon \right.$$  \hspace{1cm} (C.40)

where $\mathbb{C}$ is a constant that does not depend on $\theta$. Ho et al. (2020) find that it is better to train the diffusion model without the weighting term. The final objective is:

$$\mathbb{E}_{n \sim [1, N-1], x_0, \epsilon} \left[ \left| \epsilon - \epsilon_{\theta} \left( \sqrt{\alpha_{n+1}} x_0 + \sqrt{1 - \alpha_{n+1}} \epsilon, n + 1 \right) \right| \right]$$  \hspace{1cm} (C.41)

D. Proof of proportionality

Here we provide the proof of Theorem 4.1 that $\hat{q}(x_n \mid x_{n+1}, y)$ is proportional to $q(x_n \mid x_{n+1}) \hat{q}(y \mid x_n)$. We first define the Markov process $\hat{q}$ which is conditioned on label $y$:

$$\hat{q}(x_0) := q(x_0), \quad \hat{q}(x_1; x_0, y) := \prod_{n=1}^{N} \hat{q}(x_n \mid x_{n-1}, y), \quad \text{D.44}$$

$$\hat{q}(x_{n+1} \mid x_n, y) := \mathcal{N}(x_n; \sqrt{1 - \beta_n} x_{n-1}, \beta_n \mathbb{I}) = q(x_{n+1} \mid x_n),$$

$$\hat{q}(y \mid x_0) := \text{a label distribution on } x_0 \text{ which is known.}$$

Next, we prove that the conditional process $\hat{q}$ has the following properties.
Lemma D.1. The conditional process $\hat{q}$ has the following properties:

\[
\hat{q} (x_{n+1} | x_n) = q (x_{n+1} | x_n) \tag{D.45}
\]

\[
\hat{q} (x_{1:N} | x_0) = q (x_{1:N} | x_0) \tag{D.46}
\]

\[
\hat{q} (x_n) = q (x_n) \tag{D.47}
\]

\[
\hat{q} (x_n | x_{n+1}) = q (x_n | x_{n+1}) \tag{D.48}
\]

\[
\hat{q} (y | x_n, x_{n+1}) = \hat{q} (y | x_n) \tag{D.49}
\]

Proof. We first prove Equation (D.45):

\[
\hat{q} (x_{n+1} | x_n) = \int_y \hat{q} (x_{n+1}, y | x_n) \, dy \tag{D.50}
\]

\[
= \int_y \hat{q} (x_{n+1} | x_n, y) \hat{q} (y | x_n) \, dy \tag{D.51}
\]

\[
= \int_q (x_{n+1} | x_n) \hat{q} (y | x_n) \, dy \tag{D.52}
\]

\[
= q (x_{n+1} | x_n) \int_y \hat{q} (y | x_n) \, dy \tag{D.53}
\]

\[
= q (x_{n+1} | x_n) \tag{D.54}
\]

Following the similar proof with (D.45), we can prove Equation (D.46):

\[
\hat{q} (x_{1:N} | x_0) = \int_y \hat{q} (x_{1:N}, y | x_0) \, dy \tag{D.55}
\]

\[
= \int_y \hat{q} (y | x_0) \hat{q} (x_{1:N} | x_0, y) \, dy \tag{D.56}
\]

\[
= \int_y \hat{q} (y | x_0) \prod_{n=1}^N \hat{q} (x_n | x_{n-1}, y) \, dy \tag{D.57}
\]

\[
= \int_y \hat{q} (y | x_0) \prod_{n=1}^N q (x_n | x_{n-1}) \, dy \tag{D.58}
\]

\[
= \prod_{n=1}^N q (x_n | x_{n-1}) \int_y \hat{q} (y | x_0) \, dy \tag{D.59}
\]

\[
= \prod_{n=1}^N q (x_n | x_{n-1}) \tag{D.60}
\]

\[
= q (x_{1:N} | x_0) \tag{D.61}
\]

where Equation (D.57) uses the definition in (D.44) to derive Equation (D.58). By using Equation (D.46), we can prove Equation (D.47):

\[
\hat{q} (x_n) = \int_{x_{0:n-1}} \hat{q} (x_0, \ldots, x_n) \, dx_{0:n-1} \tag{D.62}
\]

\[
= \int_{x_{0:n-1}} \hat{q} (x_0) \hat{q} (x_1, \ldots, x_n | x_0) \, dx_{0:n-1} \tag{D.63}
\]

\[
= \int_{x_{0:n-1}} q (x_0) q (x_1, \ldots, x_n | x_0) \, dx_{0:n-1} \tag{D.64}
\]

\[
= \int_{x_{0:n-1}} q (x_0, \ldots, x_n) \, dx_{0:n-1} \tag{D.65}
\]

\[
= q (x_n) \tag{D.66}
\]
Combining (D.45) with (D.47), we can directly use Bayes’ rule to prove Equation (D.48):

\[ \hat{q}(x_n | x_{n+1}) = \hat{q}(x_{n+1} | x_n) \hat{q}(x_n) / \hat{q}(x_{n+1}) \]  
\[ = q(x_n | x_{n+1}) q(x_n) / q(x_{n+1}) \]  
\[ = q(x_n | x_{n+1}) \]  
\[ \text{(D.67)} \]

By using the definition \( \hat{q}(x_{n+1} | x_n, y) = q(x_{n+1} | x_n) \) in (D.44) and Equation (D.48), we can prove Equation (D.49):

\[ \hat{q}(y | x_n, x_{n+1}) = \hat{q}(x_{n+1} | x_n, y) \hat{q}(y | x_n) / \hat{q}(x_{n+1} | x_n) \]  
\[ = q(x_{n+1} | x_n) \hat{q}(y | x_n) / \hat{q}(x_{n+1} | x_n) \]  
\[ = \hat{q}(y | x_n) \]  
\[ \text{(D.70)} \]

Finally, by combining (D.48) and (D.49) in Lemma D.1, we can track the conditional probability \( \hat{q}(x_n | x_{n+1}, y) \):

\[ \hat{q}(x_n | x_{n+1}, y) = \frac{\hat{q}(x_n, x_{n+1}, y)}{\hat{q}(x_{n+1}, y)} \]  
\[ = \frac{\hat{q}(x_n, x_{n+1}, y)}{\hat{q}(y | x_{n+1}) \hat{q}(x_{n+1})} \]  
\[ = \frac{\hat{q}(x_n, x_{n+1}) \hat{q}(y | x_n, x_{n+1}) \hat{q}(x_{n+1})}{\hat{q}(y | x_{n+1}) \hat{q}(x_{n+1})} \]  
\[ = \frac{\hat{q}(x_n, x_{n+1}) \hat{q}(y | x_n, x_{n+1})}{\hat{q}(y | x_{n+1}) \hat{q}(x_{n+1})} \]  
\[ = \frac{\hat{q}(x_n, x_{n+1}) \hat{q}(y | x_n)}{\hat{q}(y | x_{n+1}) \hat{q}(x_{n+1})} \]  
\[ = \frac{q(x_n | x_{n+1}) \hat{q}(y | x_n)}{\hat{q}(y | x_{n+1}) \hat{q}(x_{n+1})} \]  
\[ \text{(D.74)} \]

As \( \hat{q}(y | x_{n+1}) \) is independent from \( x_n \), it can be regarded as a constant. Therefore, \( \hat{q}(x_n | x_{n+1}, y) \) is proportional to \( q(x_n | x_{n+1}) \hat{q}(y | x_n) \). Dhariwal & Nichol (2021) also provide a similar proof which can also be referred to.
E. Samples generated by DDGR

Here we present the samples generated by DDGR at the last three batches. As illustrated in Figure E.1, the samples generated by DGR are still clear.

![Samples generated by DDGR](image)

*Figure E.1. The samples generated by DDGR at the last three batches.*

F. Experiments without a large first task

To further demonstrate the advantages of DDGR, we try to conduct experiments on CIFAR100 without a large first task by AlexNet. We split CIFAR100 into five tasks, and each task has 20 classes. The chosen baseline is PASS which outputs other baselines in Section 5.2. The results are presented in Table F.2.

<table>
<thead>
<tr>
<th></th>
<th>Phase 0</th>
<th>Phase 1</th>
<th>Phase 2</th>
<th>Phase 3</th>
<th>Phase 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS</td>
<td>70.51</td>
<td>52.17</td>
<td>50.41</td>
<td>47.39</td>
<td>44.32</td>
</tr>
<tr>
<td>DDGR</td>
<td>70.65</td>
<td>54.33</td>
<td>53.35</td>
<td>51.16</td>
<td>49.03</td>
</tr>
</tbody>
</table>

As shown in Table F.2, DDGR also achieves the best performance in all experiments. For example, DDGR outperforms the best baseline by 4.71% in terms of the final average accuracy. The average accuracy of DDGR is 0.14% to 4.71% higher than that of the best baseline at each phase. Therefore, DDGR still works without a large first task.

G. The effect of diffusion steps

To figure out the effect of diffusion steps, we conduct experiments and present the results with numbers of different diffusion steps in Table G.3. Overall, the gap of the final average accuracy and forgetting rate between different $N$ is small. For example, when $NC$ (number of classes in the incremental tasks) is 5, the final average accuracy ranges from 56.66% to 59.20%, representing a 2.54% difference.
Table G.3. We present the final average accuracy $A_T$ and forgetting rate $F_T^{avg}$ on CIFAR100 with different numbers of diffusion model steps. The classifier is AlexNet.

<table>
<thead>
<tr>
<th>$N$</th>
<th>NC=5</th>
<th>10</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>56.66</td>
<td>50.20</td>
<td>24.83</td>
<td>19.43</td>
</tr>
<tr>
<td>2000</td>
<td>59.02</td>
<td>52.78</td>
<td>22.78</td>
<td>15.10</td>
</tr>
<tr>
<td>3000</td>
<td>56.80</td>
<td>51.10</td>
<td>24.30</td>
<td>19.52</td>
</tr>
<tr>
<td>4000</td>
<td>59.20</td>
<td>52.22</td>
<td>23.00</td>
<td>16.86</td>
</tr>
</tbody>
</table>

H. Time cost of DDGR

We present the time cost in Table H.4, which indicates that DDGR should be improved in terms of the time cost.

Table H.4. The total time cost (s) of baselines on CIFAR100.

<table>
<thead>
<tr>
<th>method</th>
<th>$NC=5$</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finetuning</td>
<td>2421.40</td>
<td>3541.63</td>
</tr>
<tr>
<td>SI</td>
<td>3638.51</td>
<td>5167.41</td>
</tr>
<tr>
<td>EWC</td>
<td>5229.43</td>
<td>7114.12</td>
</tr>
<tr>
<td>MAS</td>
<td>6923.60</td>
<td>8491.71</td>
</tr>
<tr>
<td>IMM</td>
<td>8054.56</td>
<td>10324.09</td>
</tr>
<tr>
<td>DGR</td>
<td>69532.44</td>
<td>72543.57</td>
</tr>
<tr>
<td>DDGR</td>
<td>75692.41</td>
<td>87514.23</td>
</tr>
</tbody>
</table>

We count the time cost of all experiments on CIFAR100. Generally speaking, DDGR spends more time than other baselines. In Table H.5, we further analyse the time cost of DDGR, and find that the training and sampling process of diffusion model cost too much time while the calculation of instruction-operator only spends a little time. These indicate that DDGR spends a lot of time due to the training and sampling process of diffusion model rather than the calculation of our proposed instruction-operator.

Table H.5. The every part of time cost (s) of DDGR on CIFAR100.

<table>
<thead>
<tr>
<th>$NC=5$</th>
<th>$NC=10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 0</td>
<td>Phase 1</td>
</tr>
<tr>
<td>Instruction-operator (IO)</td>
<td>0.00</td>
</tr>
<tr>
<td>Sampling (except IO)</td>
<td>0.00</td>
</tr>
<tr>
<td>Classifier training</td>
<td>223.93</td>
</tr>
<tr>
<td>Diffusion model training</td>
<td>12776.03</td>
</tr>
<tr>
<td>Sum</td>
<td>12999.96</td>
</tr>
</tbody>
</table>