# Supplementary Material: Class-Incremental Learning with Generative Classifiers

Gido M. van de Ven[1,2,*], Zhe Li[1] & Andreas S. Tolias[1,3]

[1] Center for Neuroscience and Artificial Intelligence, Baylor College of Medicine, Houston, Texas, USA

[2] Computational and Biological Learning Lab, University of Cambridge, Cambridge, United Kingdom

[3] Department of Electrical and Computer Engineering, Rice University, Houston, Texas, USA

## A. Experimental details

Documented PyTorch code to perform and build upon the experiments described in this paper is available online: https://github.com/GMvandeVen/class-incremental-learning.

### A.1. Technical details of the compared methods

This section provides the technical details of the methods compared against in Table 2 in the main text. Most of the compared methods use deep neural networks that are trained by minimizing the multi-class cross-entropy loss (although some methods have additional terms in the loss function, see below) given by:

$$\mathcal{L}^{\text{CE}}\left(\boldsymbol{\theta};\boldsymbol{x},y\right) = -\log p_{\boldsymbol{\theta}}\left(Y=y|\boldsymbol{x}\right) \tag{1}$$

whereby $p_{\boldsymbol{\theta}}\left(Y=y|\boldsymbol{x}\right)$ is the probability that input $\boldsymbol{x}$ belongs to class $y$ as predicted by the neural network with parameters $\boldsymbol{\theta}$. These probabilities are computed by performing a softmax normalization on the activations of the final output layer of the network. It is important to note that this softmax normalization is only performed over the output units of the classes that have been seen by the network up to that point in time. In other words, the networks are trained with an *"expanding head"* [13] or only the classes seen so far are set as *"active"* [17]. In this Appendix, this way of training a deep neural network is referred to as the "standard way".

In addition to the methods discussed below, the performance of the following two baselines is reported in Table 2 in the main text:

**- None:** The base neural network is sequentially trained on all tasks in the standard way. This baseline suffers from severe catastrophic forgetting, and is included as a lower bound.

**- Joint:** The base neural network is trained on all classes at the same time. For this baseline the same total number of iterations is used as for the incremental training protocols, with the difference that in each iteration the training data is randomly sampled from all classes rather than just from the classes in the current task. This baseline can be seen as an upper bound.

#### A.1.1 Deep generative replay

With deep generative replay [DGR], following [14], a separate generative model is trained to generate input samples to be replayed. We use a variational autoencoder [VAE; 8] as generator. The encoder network of the VAE is always similar to the base network (except for the final softmax layer) and the decoder network is the mirror image of the encoder network; see Section A.3 for the exact VAE architectures used for each benchmark. The classifier, or main model, is simply the base neural network.

Except on the first task, both the classifier and the generator are trained with replay. The replay is generated by sampling inputs from a copy of the generator, after which those inputs are labelled as the most likely class as predicted by a copy the classifier. The versions of the generator and classifier used to produce the replay are temporarily stored copies of both models after finishing training on the previous task.

Following [16], for both the classifier and the generator, the total loss is a weighted sum of the loss on the data from the current task and the loss on the replayed data: $\mathcal{L} = \frac{1}{N_{\text{tasks so far}}}\mathcal{L}_{\text{current}} + (1-\frac{1}{N_{\text{tasks so far}}})\mathcal{L}_{\text{replay}}$. For the main model, $\mathcal{L}_{\text{current}}$ and $\mathcal{L}_{\text{replay}}$ are the cross-entropy loss (see Eq. 1). For the generator, $\mathcal{L}_{\text{current}}$ and $\mathcal{L}_{\text{replay}}$ are the VAE loss (see Eq. 13). In each iteration, the number of replayed samples is equal to the number of samples from the current task.

#### A.1.2 Brain-inspired replay

For brain-inspired replay [BI-R] we follow the exact protocol as described in [15], using the code released by the au-

*Corresponding author: ven@bcm.edu

thors. We use all five of the proposed modifications (distillation, replay-through-feedback, conditional replay, gating based on internal context and internal replay). Because the internal replay component of BI-R relies on the availability of a pre-trained feature extractor, we do not use BI-R on the MNIST and CIFAR-10 benchmarks. Note that BI-R has a hyperparameter $X$, which controls the percentage of hidden units in the decoder that is masked per class (see Section A.5).

### A.1.3 EWC & SI

For elastic weight consolidation [EWC; 9] and synaptic intelligence [SI; 18], the base neural network is trained in the standard way, except that a regularization term is added to the cross-entropy loss: $\mathcal{L} = \mathcal{L}^{\text{CE}} + \lambda\mathcal{L}^{\text{REG}}$, whereby hyperparameter $\lambda$ controls the regularization strength (see Section A.5). This regularization term penalizes changes to parameters important for previously learned tasks.

**EWC** The regularization term for EWC is given by:

$$\mathcal{L}^{\text{REG}}(\boldsymbol{\theta}) = \sum_{k=1}^{K-1}\left(\frac{1}{2}\sum_{i=1}^{N_{\text{params}}} F_{ii}^{(k)}\left(\theta_i - \hat{\theta}_i^{(k)}\right)^2\right) \tag{2}$$

whereby $K$ is the current task, $\hat{\theta}_i^{(k)}$ are the parameters of the network after training on task $k$ and $F_{ii}^{(k)}$ is the estimated importance of parameter $i$ for task $k$. This last one is calculated as the $i^{\text{th}}$ diagonal element of the Fisher Information matrix of task $k$:

$$F_{ii}^{(k)} = E_{\boldsymbol{x}\sim S^{(k)}}\left[\sum_{c=1}^{N_{\text{classes}}} \tilde{y}_c^{(\mathbf{x})}\left(\frac{\delta \log p_{\boldsymbol{\theta}}(Y=c|\mathbf{x})}{\delta\theta_i}\bigg|_{\hat{\boldsymbol{\theta}}^{(k)}}\right)^2\right] \tag{3}$$

whereby $S^{(k)}$ is the training data of task $k$ and $\tilde{y}_c^{(\mathbf{x})} = p_{\hat{\boldsymbol{\theta}}^{(k)}}(Y=c|\mathbf{x})$.

**SI** The regularization term for SI is given by:

$$\mathcal{L}^{\text{REG}}(\boldsymbol{\theta}) = \sum_{i=1}^{N_{\text{params}}} \Omega_i^{(K-1)}\left(\theta_i - \hat{\theta}_i^{(K-1)}\right)^2 \tag{4}$$

whereby $K$ is the current task, $\hat{\theta}_i^{(K-1)}$ are the parameters of the network after training on task $K-1$ and $\Omega_i^{(K-1)}$ is the estimated importance of parameter $i$ for the first $K-1$ tasks. To calculate these $\Omega_i^{(K-1)}$, after each task $k$, a per-parameter contribution to the change in loss is computed:

$$\omega_i^{(k)} = \sum_{t=1}^{N_{\text{iters}}^k}\left(\hat{\theta}_i[t^{(k)}] - \hat{\theta}_i[(t-1)^{(k)}]\right)\frac{-\delta\mathcal{L}_{\text{total}}[t^{(k)}]}{\delta\theta_i} \tag{5}$$

whereby $N_{\text{iters}}^k$ is the number of iterations for task $k$, $\hat{\theta}_i[t^{(k)}]$ is the value of the $i^{\text{th}}$ parameter after the $t^{\text{th}}$ training iteration on task $k$ and $\frac{\delta\mathcal{L}_{\text{total}}[t^{(k)}]}{\delta\theta_i}$ is the gradient of the loss with respect to the $i^{\text{th}}$ parameter during the $t^{\text{th}}$ training iteration on task $k$. The $\Omega_i^{(K-1)}$ are then calculated as:

$$\Omega_i^{(K-1)} = \sum_{k=1}^{K-1}\frac{\omega_i^{(k)}}{\left(\Delta_i^{(k)}\right)^2 + \xi} \tag{6}$$

whereby $\xi$ is a dampening term that was set to 0.1 and $\Delta_i^{(k)} = \hat{\theta}_i[N_{\text{iters}}^{k\ (k)}] - \hat{\theta}_i[0^{(k)}]$, where $\hat{\theta}_i[0^{(k)}]$ is the value of parameter $i$ when training on task $k$ started.

### A.1.4 CWR, CWR+ & AR1

**CWR** For the method 'CopyWeights with Re-init' [CWR; 10], the base neural network is trained on the first task in the standard way. After the first task, all parameters of the network are frozen except for the parameters of the output layer. For the parameters of the output layer, two copies are maintained: a temporary version denoted **tw**, and a 'consolidated' version denoted **cw**. Training is done with **tw**. Before starting training on each task, **tw** is randomly re-initialized. After finishing training on each task, the parameters in **tw** corresponding to the classes of that task are copied over into **cw**. For testing, **cw** is used.

**CWR+** An improved version of CWR, called CWR+, was proposed in [13]. CWR+ has two differences compared to CWR. First, before each task, the parameters in **tw** are set to zero rather than randomly (re-)initialized. Second, after each task, the parameters in **tw** are first standardized by subtracting their mean (with the mean taken over all classes seen up to that point, and with a separate mean for the weights and the biases), and then the standardized parameters corresponding to the classes of that task are copied over into **cw**.

**AR1** Building upon CWR+, Maltoni & Lomonaco [13] also proposed AR1. This method is similar to CWR+, except that the parameters of the hidden layers are not frozen after the first task. Instead, a modified version of SI is used. The first modification is that SI is only used for the parameters of the hidden layers and not for the parameters of the output layer. The second modification is that $\Omega_i^{(K-1)}$ in Eq. 4 is replaced by $\tilde{\Omega}_i^{(K-1)} = \max\left\{\Omega_i^{(K-1)}, \Omega_{\text{max}}\right\}$, with $\Omega_{\text{max}}$ a newly introduced hyperparameter that limits the extent to which each parameter could be regularized (see Section A.5).

### A.1.5 Labels trick

For the 'labels trick' [19], the base network is trained in the standard way, except that always only the classes from the current task are set as 'active' (see first paragraph of Section A.1). This means that the softmax normalization is only performed over the output units of those classes, and that the network is therefore only trained on the classes from the current task. Another way to phrase this is that the network is trained as if it is trained on a task-incremental learning problem [17]. A fundamental limitation of this trick is that the network is never trained to learn to distinguish between classes from different tasks.

### A.1.6 SLDA

The method streaming linear discriminant analysis [SLDA; 4] learns a linear classifier of the form:

$$\hat{y} = \underset{c \in \mathcal{Y}}{\operatorname{argmax}} \left\{ \boldsymbol{w}_c^{\mathrm{T}} \boldsymbol{x} + b_c \right\} \tag{7}$$

whereby $\boldsymbol{w}_c$ is the $c^{\text{th}}$ row of weight matrix $\mathbf{W}$, $b_c$ is the $c^{\text{th}}$ element of bias vector $\boldsymbol{b}$ and $\hat{y}$ is the predicted class label.

To learn $\mathbf{W}$ and $\boldsymbol{b}$, SLDA computes for each class $y$ a mean vector $\boldsymbol{\mu}_y$ and associated count $n_y$, as well as a single covariance matrix $\boldsymbol{\Sigma}$ that is shared between all classes. Updates to $\boldsymbol{\mu}_y$ and $n_y$ are done in a "pure streaming" manner: they are initialized at zero and for each new training sample $(\boldsymbol{x}, y)$ that arrived at time $t$, they are updated as:

$$\boldsymbol{\mu}_y^{(t+1)} = \frac{n_y^{(t)} \boldsymbol{\mu}_y^{(t)} + \boldsymbol{x}}{n_y^{(t)} + 1} \tag{8}$$

$$n_y^{(t+1)} = n_y^{(t)} + 1 \tag{9}$$

with $\boldsymbol{\mu}_y^{(t)}$ and $n_y^{(t)}$ the versions of $\boldsymbol{\mu}_y$ and $n_y$ at time $t$. The covariance matrix is initialized on the first task using the Oracle Approximating Shrinkage estimator [2], which is a batch-wise computation. On subsequent tasks, updates to $\boldsymbol{\Sigma}$ are done in a streaming manner: for each new training sample $(\boldsymbol{x}, y)$ that arrives at time $t$, the following update is done:

$$\boldsymbol{\Sigma}^{(t+1)} = \frac{t\boldsymbol{\Sigma}^{(t)} + \boldsymbol{\Delta}^{(t)}}{t+1} \tag{10}$$

with $\boldsymbol{\Delta}^{(t)} = \frac{t}{t+1}\left(\boldsymbol{x} - \boldsymbol{\mu}_y^{(t)}\right)\left(\boldsymbol{x} - \boldsymbol{\mu}_y^{(t)}\right)^{\mathrm{T}}$ and $\boldsymbol{\Sigma}^{(t)}$ the version of $\boldsymbol{\Sigma}$ at time $t$.

To perform classification, the rows of $\mathbf{W}$ and the elements of $\boldsymbol{b}$ are then computed as:

$$\boldsymbol{w}_c = \boldsymbol{\Lambda} \boldsymbol{\mu}_c \tag{11}$$

$$b_c = \boldsymbol{\mu}_c^{\mathrm{T}} \boldsymbol{\Lambda} \boldsymbol{\mu}_c \tag{12}$$

where $\boldsymbol{\Lambda} = \left[(1-\epsilon)\boldsymbol{\Sigma} + \boldsymbol{\epsilon}\mathbf{I}\right]^{-1}$ and $\epsilon = 0.0001$.

With SLDA it is not possible to train the parameters of a deep neural network. However, as pointed out in [4], it is possible to use a pre-trained deep neural network as feature extractor. On the CIFAR-100 and CORe50 benchmarks, we use the pre-trained networks that are available for those benchmarks as feature extractor. On the MNIST and CIFAR-10 benchmarks, for which no pre-trained networks are available, we apply SLDA directly on the raw inputs.

## A.2. VAE training

VAE models are trained both for the generative classifier and for the generative replay variants. In both cases, the set-up of the VAE models and their training is similar, except that the VAE models of the generative classifier are smaller than those used for generative replay (see Section A.3).

Each VAE model consists of two deep neural networks: (1) an encoder network, parameterized by $\boldsymbol{\phi}$, mapping an input $\boldsymbol{x}$ to the mean $\boldsymbol{\mu}_{\boldsymbol{\phi}}^{(\boldsymbol{x})}$ and standard deviation $\boldsymbol{\sigma}_{\boldsymbol{\phi}}^{(\boldsymbol{x})}$ of the posterior distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}\left(\boldsymbol{z} \,\middle|\, \boldsymbol{\mu}_{\boldsymbol{\phi}}^{(\boldsymbol{x})}, {\boldsymbol{\sigma}_{\boldsymbol{\phi}}^{(\boldsymbol{x})}}^2 I\right)$ over the latent variables $\boldsymbol{z}$; and (2) a decoder network, parameterized by $\boldsymbol{\phi}$, mapping a latent variable vector $\boldsymbol{z}$ to a reconstructed input $\boldsymbol{\mu}_{\boldsymbol{\theta}}^{(\boldsymbol{z})}$, which is used as the mean of a Gaussian observer model $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) = \mathcal{N}\left(\boldsymbol{x} \,\middle|\, \boldsymbol{\mu}_{\boldsymbol{\theta}}^{(\boldsymbol{z})}, I\right)$. The prior distribution over the latent variables $\boldsymbol{z}$ is the standard normal distribution: $p_{\text{prior}}(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z} \,|\, \mathbf{0}, I)$.

The parameters of these two networks are trained by maximizing a variational lower bound to the likelihood, or ELBO (see Eq. 4 in the main text), which is equivalent to minimizing the following loss function:

$$\begin{aligned}
\mathcal{L}^{\text{VAE}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) &= E_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[-\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&= E_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[-\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})] + D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) || p_{\text{prior}}(\boldsymbol{z})) \\
&= \mathcal{L}^{\text{recon}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) + \mathcal{L}^{\text{latent}}(\boldsymbol{\phi}; \boldsymbol{x})
\end{aligned} \tag{13}$$

whereby $D_{KL}$ is the Kullback-Leibler divergence. The first term in this loss function can be simplified to:

$$\mathcal{L}^{\text{recon}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = E_{\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})}\left[\sum_{i=1}^{N_{\text{inputs}}} \left(x_i - \mu_{\tilde{\boldsymbol{z}}, i}\right)^2\right] \tag{14}$$

whereby $x_i$ is the $i^{\text{th}}$ element of the original input $\boldsymbol{x}$ and $\mu_{\tilde{\boldsymbol{z}}, i}$ is the $i^{\text{th}}$ element of the decoded input $\boldsymbol{\mu}_{\tilde{\boldsymbol{z}}}$, with $\tilde{\boldsymbol{z}} = \boldsymbol{\mu}_{\boldsymbol{\phi}}^{(\mathbf{x})} + \boldsymbol{\sigma}_{\boldsymbol{\phi}}^{(\mathbf{x})} \odot \boldsymbol{\epsilon}$. We estimate Eq. 14 with a single sample of $\boldsymbol{\epsilon}$ for each datapoint.

The second term in Eq. 13 is calculated analytically[1]:

$$\mathcal{L}^{\text{latent}}(\boldsymbol{\phi};\boldsymbol{x}) = \frac{1}{2}\sum_{j=1}^{N_{\text{latent}}}\left(1+\log(\sigma_{\boldsymbol{\phi},j}^{(\mathbf{x})\,2}) - \mu_{\boldsymbol{\phi},j}^{(\mathbf{x})\,2} - \sigma_{\boldsymbol{\phi},j}^{(\mathbf{x})\,2}\right) \tag{15}$$

where $\mu_{\boldsymbol{\phi},j}^{(\mathbf{x})}$ and $\sigma_{\boldsymbol{\phi},j}^{(\mathbf{x})}$ are the $j^{\text{th}}$ elements of $\boldsymbol{\mu}_{\boldsymbol{\phi}}^{(\mathbf{x})}$ and $\boldsymbol{\sigma}_{\boldsymbol{\phi}}^{(\mathbf{x})}$, and $N_{\text{latent}}$ is the dimension of the latent space.

## A.3. Architectures & training settings

Neural network training is always done using the Adam-optimizer [7] with default settings (*i.e.* $\beta_1 = 0.9$, $\beta_2 = 0.999$). Depending on the benchmark, the learning rate is 0.001 (MNIST and CIFAR-10) or 0.0001 (CIFAR-100 and CORe50).

### A.3.1 MNIST

For the MNIST benchmark, the base neural network has two fully-connected hidden layers with 400 ReLU units each, followed by a softmax output layer.

For DGR, the generative model is a symmetric VAE with both the encoder network and the decoder network similar to the base network (*i.e.* two fully-connected layers with 400 units each). The dimension of the latent space is 100. The same architecture was used in previous studies [6, 17].

For our generative classifier implementation, we use VAE models with both the encoder network and the decoder network consisting of two fully-connected layers with 85 units. The dimension of the latent space is 5.

### A.3.2 CIFAR-10

For the CIFAR-10 benchmark, following several previous studies [1, 3, 12], the base neural network is a slimmed down version of ResNet18 [5]. In each layer, this version has approximately three times less channels than the the standard ResNet18: it has 20, 20, 40, 80 and 160 channels in the subsequent layers (instead of 64, 64, 128, 256 and 512). After the final residual block, global average pooling is applied, which is followed by the softmax output layer.

For DGR, the generative model is a VAE whereby the encoder network is similar to the base neural network (except that no pooling is used and there is no softmax output layer) and the decoder network is the mirror image of the encoder network. The dimension of the latent space is 100.

The VAE models that are used for the generative classifier have an encoder network that consists of three standard convolutional layers (with 15, 30 and 60 channels; each layer used batchnorm, ReLU non-linearities, a 3x3 kernel, a padding of 1 and a stride of 2), a decoder network that is the mirror image of the encoder network and a latent space of dimension 100.

[1] See Appendix B in [8] for the full derivation.

### A.3.3 CIFAR-100

For the CIFAR-100 benchmark, following [15], the base neural network has five pre-trained convolutional layers (16, 32, 64, 128 and 254 channels) followed by two randomly-initialized fully-connected layers with 2000 ReLU units and a softmax output layer. The convolutional layers are the same ones as in [15]: they use batch-norm, ReLU non-linearities, a 3x3 kernel, a padding of 1, and a stride of 1 (first layer) or 2 (all other layers). They have been pre-trained on CIFAR-10 for 100 epochs using the ADAM-optimizer ($\beta_1 = 0.9$, $\beta_2 = 0.999$), a learning rate of 0.0001 and a mini-batch size of 256. On this benchmark, all methods are run twice: once with the pre-trained convolutional layers frozen and once with those layers plastic. Reported for each method in Table 2 in the main text is the variant that performed best. For all methods this is the variant with the convolutional layers frozen, except for AR1 and the joint training baseline.

The generative model for DGR is a symmetric VAE with as encoder network the base neural network, as decoder network a mirror image of the encoder network and latent space dimension of 100. For BI-R, the combined classifier/generator model is the same as the VAE for DGR, except that the deconvolutional layers are removed from the decoder network and that a softmax output layer is appended to the top layer of the encoder network.

For the generative classifier, reminiscent of the approach of BI-R, we train the VAE models on the features extracted by the pre-trained convolutional layers rather than on the raw inputs. That means that the reconstruction loss of the VAE models is in the feature space instead of at the pixel level. The VAE models that we use have an encoder network and a decoder network both consisting of one fully-connected hidden layer with 85 ReLU units and a latent space with dimension 20.

### A.3.4 CORe50

For the CORe50 benchmark, the base neural network is a standard ResNet18 that has been pre-trained on ImageNet (downloaded from PyTorch), followed by one fully-connected layer with 1024 ReLU units and a softmax output layer. For all methods, the parameters of the ResNet18 are frozen, and only the fully-connected layer and the output layer are trained.

On this benchmark we do not perform DGR, because we do not believe that training a pixel-level generative model in a pure streaming manner on CORe50 stands any chance of success (except perhaps if the full generative model has been pre-trained). For BI-R, the reconstruction objective is placed at the level of the features extracted by the pre-trained ResNet18. The model used for BI-R has an encoder network and a decoder network that both consist of

Table A.1. Overview of the explored and selected hyperparameter values.

| Method | Param | Explored range | Selected values | | | |
|---|---|---|---|---|---|---|
| | | | MNIST | CIFAR-10 | CIFAR-100 | CORe50 |
| BI-R | $X$ | $[0, 10, 20, ..., 80, 90]$ | - | - | 70 | 0 |
| BI-R + SI | $X$ | $[0, 20, 40, 60, 80]$ | - | - | 60 | 60 |
| | $\lambda$ | $[0, 0.001, 0.01, ..., 10^8, 10^9]$ | - | - | $10^8$ | 0.01 |
| SI | $\lambda$ | $[0, 0.001, 0.01, ..., 10^8, 10^9]$ | $10^3$ | 1 | 1 | 10 |
| EWC | $\lambda$ | $[0, 0.1, 1, ..., 10^6, 10^7]$ | $10^6$ | 10 | 100 | 10 |
| AR1 | $\lambda$ | $[0, 0.001, 0.01, ..., 10^8, 10^9]$ | 10 | 100 | 100 | 1 |
| | $\Omega_{\text{max}}$ | $[0.0001, 0.001, ..., 10, 100]$ | 0.01 | 0.1 | 10 | 0.1 |

one fully-connected hidden layer with 1024 ReLU units, it has a softmax output layer on top of the encoder network and the dimension of the latent space is 200.

The VAE models of the generative classifier are trained on the features extracted by the pre-trained ResNet18. These VAE models have no hidden layers (*i.e*. there is only a fully-connected layer from the ResNet embeddings to the latent space) and a latent space of dimension 110.

### A.4. Direct comparison between generative classification and generative replay

For the experiments described in Section 5.3 in the main text, a softmax-based classifier is trained in an i.i.d. manner on samples generated by the VAE models of the generative classifier. The classifier used for these experiments is the base neural network of each benchmark. This network is trained for the same number of iterations (and using the same mini-batch size and training settings) as for the joint training baseline. The only difference with the joint training baseline is that each mini-batch is made up of samples generated by the VAE models rather than by samples from the original training data. The samples are generated by first randomly sampling a class from all possible classes, after which a sample is drawn from the VAE model of that class.

### A.5. Hyperparameter searches

Several of the methods we compare against have one or more hyperparameters. Hyperparameters in continual learning can be problematic, because typically they are set by running a method on the full benchmark with a range of different hyperparameter-values. This means that these parameters are 'learned' in a non-continual way, see also the discussion in the Appendix of [17]. Nevertheless, to give the methods we compare against the best chance, we select their hyperparameters based on gridsearches (see Table A.1). These gridsearches are performed with a single random seed. The results in Table 2 in the main text are then obtained with ten different random seeds.

## B. A further distinction: batch-wise *vs.* streaming

Within task-based class-incremental learning, a further distinction can be made depending on whether within each task the algorithm is given free access to all data at once ("task-based batch-wise") or whether the task's data is fed to the algorithm according to a fixed stream outside of the control of the algorithm ("task-based streaming"). This distinction is important because some continual learning methods perform at each task boundary a consolidation operation that requires cycling over the training data of that task (*e.g*. estimating the Fisher Information matrix in EWC), so those methods are only suitable for task-based batch-wise learning. Another difference is that with batch-wise learning, training settings such as number of iterations and mini-batch size can be decided on by the algorithm itself, while in streaming learning these are part of the benchmark.

Table B.1 provides an overview of which methods can be applied in different class-incremental learning settings:

- The generative replay methods (DGR and BI-R) require task boundaries in order to decide when to update the copy of the models used to generate the replay. The current version of these methods are therefore not suitable for task-free continual learning. Neither DGR or BI-R makes assumptions about the way the data within each task is encountered, so both methods can be used for task-based streaming learning.

- The regularization-based methods (EWC and SI) require task boundaries in order to decide when to update their regularization term, and the standard version of these methods are therefore not suitable for task-free continual learning (although see [19] for a possible work around). EWC additionally makes the assumption that at each task boundary it is possible to make another pass over the training data of that task to estimate the Fisher Information matrix. EWC can there-

Table B.1. Overview of class-incremental learning variants to which the methods that are compared in this paper can be applied to. The symbols $^{(+)}$ and $^{(*)}$ indicate nuances that are discussed in Section B.

| **Strategy** | **Method** | ***Task-based batch-wise*** | ***Task-based streaming*** | ***Task-free streaming*** |
|---|---|---|---|---|
| Generative Replay | DGR | v | v | - |
| | BI-R | v | v | - |
| Regularization | EWC | v | - | - |
| | SI | v | v | - |
| Bias-correction | CWR / CWR+ | v | v | v$^{(+)}$ |
| | AR1 | v | v | v$^{(+)}$ |
| | Labels Trick | v | v | - |
| Other | SLDA | v | v$^{(*)}$ | v$^{(*)}$ |
| Generative Classifier | | v | v | v |

fore only be applied to task-based batch-wise learning, while SI can also be used for task-based streaming learning.

- The bias correction methods (CWR, CWR+, AR1 and the labels trick) do not make any assumptions about how the data within each task is encountered, and all of them are applicable to both variants of task-based learning. The labels trick relies on specified tasks for the task-specific training, so this approach is not suitable for the task-free setting. The original versions of CWR, CWR+ and AR1 are also not compatible with task-free continual learning because they rely on the task boundaries for consolidating the (task-specific) weights of the output layer, but recently an adaption of these methods has been proposed to make them suitable for the task-free setting [11].

- In principle, SLDA is a streaming method that is generally applicable to both task-based and task-free class-incremental learning. However, SLDA does make the assumption that its covariance matrix can be initialized on the first task (or in a separate 'base initialization phase') in a batch-wise operation.

- The generative classifier strategy proposed in this paper can be applied to both task-based and task-free class-incremental learning, as it does not make any assumptions about task boundaries or about being able to access larger amounts of data at the same time. In fact, for the specific implementation of the generative classifier that was considered in this paper — with a separate VAE model for each class to be learned — the specific sequence in which the different classes are presented does not matter at all, because the model for each class is trained only on data from its own class.

## References


[1] Rahaf Aljundi, Eugene Belilovsky, Tinne Tuytelaars, Laurent Charlin, Massimo Caccia, Min Lin, and Lucas Page-Caccia. Online continual learning with maximal interfered retrieval. In *Advances in Neural Information Processing Systems*, pages 11849–11860, 2019.

[2] Yilun Chen, Ami Wiesel, Yonina C Eldar, and Alfred O Hero. Shrinkage algorithms for mmse covariance estimation. *IEEE Transactions on Signal Processing*, 58(10):5016–5029, 2010.

[3] Matthias De Lange and Tinne Tuytelaars. Continual prototype evolution: Learning online from non-stationary data streams. *arXiv preprint arXiv:2009.00919*, 2020.

[4] Tyler L Hayes and Christopher Kanan. Lifelong machine learning with deep streaming linear discriminant analysis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 220–221, 2020.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[6] Yen-Chang Hsu, Yen-Cheng Liu, and Zsolt Kira. Re-evaluating continual learning scenarios: A categorization and case for strong baselines. *arXiv preprint arXiv:1810.12488*, 2018.

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[8] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[9] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, page 201611835, 2017.

[10] Vincenzo Lomonaco and Davide Maltoni. Core50: a new

dataset and benchmark for continuous object recognition. In *Conference on Robot Learning*, pages 17–26. PMLR, 2017.

[11] Vincenzo Lomonaco, Davide Maltoni, and Lorenzo Pellegrini. Rehearsal-free continual learning over small non-iid batches. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 989–998, 2020.

[12] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6470–6479, 2017.

[13] Davide Maltoni and Vincenzo Lomonaco. Continuous learning in single-incremental-task scenarios. *Neural Networks*, 116:56–73, 2019.

[14] Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. Continual learning with deep generative replay. In *Advances in Neural Information Processing Systems*, pages 2994–3003, 2017.

[15] Gido M van de Ven, Hava T Siegelmann, and Andreas S Tolias. Brain-inspired replay for continual learning with artificial neural networks. *Nature Communications*, 11:4069, 2020.

[16] Gido M van de Ven and Andreas S Tolias. Generative replay with feedback connections as a general strategy for continual learning. *arXiv preprint arXiv:1809.10635*, 2018.

[17] Gido M van de Ven and Andreas S Tolias. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*, 2019.

[18] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *Proceedings of the 34th International Conference on Machine Learning*, pages 3987–3995, 2017.

[19] Chen Zeno, Itay Golan, Elad Hoffer, and Daniel Soudry. Task agnostic continual learning using online variational bayes. *arXiv preprint arXiv:1803.10123v3*, 2019.