Conditional Channel Gated Networks for Task-Aware Continual Learning Supplementary Material

Davide Abati[1*] Jakub Tomczak[2] Tijmen Blankevoort[2] Simone Calderara[1]
Rita Cucchiara[1] Babak Ehteshami Bejnordi[2]

[1]University of Modena and Reggio Emilia
{name.surname}@unimore.it

[2]Qualcomm AI Research[†]
Qualcomm Technologies Netherlands B.V.
{jtomczak,tijmen,behtesha}@qti.qualcomm.com

1. Training details and hyperparameters

In this section we report training details and hyperparameters used for the optimization of our model. As already specified in Sec. 4.1 of the main paper, all models were trained with Stochastic Gradient Descent with momentum. Gradient clipping was utilized, ensuring the gradient magnitude to be lower than a predetermined threshold. Moreover, we employed a scheduler dividing the learning rate by a factor of 10 at certain epochs. Such details can be found, for each dataset, in Tab. 1, where we highlighted two sets of hyperparameters:

- *optim*: general optimization choices that were kept fixed both for our model and competing methods, in order to ensure fairness.
- *our*: hyperparameters that only concern our model, such as the weight of the sparsity loss and the number of epochs after which sparsity was introduced (patience).

[*]Research conducted during an internship at Qualcomm Technologies Netherlands B.V.

[†]Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc.

		Split MNIST	Split SVHN
optim	batch size	256	256
	learning rate	0.01	0.01
	momentum	0.9	0.9
	lr decay	-	$[400, 600]$
	weight decay	$5e-4$	$5e-4$
	epochs per task	400	800
	grad. clip	1	1
our	λ_s	0.5	0.5
	$\mathcal{L}_{sparse}$ patience	20	20
		Split CIFAR-10	**Imagenet-50**
optim	batch size	64	64
	learning rate	0.1	0.1
	momentum	0.9	0.9
	lr decay	$[100, 150]$	$[100, 150]$
	weight decay	$5e-4$	$5e-4$
	epochs per task	200	200
	grad. clip	1	1
our	λ_s	1	1
	$\mathcal{L}_{sparse}$ patience	10	0

Table 1: Hyperparameters table.

2. WGAN details

This section illustrates architectures and training details for the generative models employed in Sec. 4.4 of the main paper. As stated in the manuscript, we rely on the framework of Wasserstein GANs with Gradient Penalty (WGAN-GP, [2]). The reader can find the specification of the architecture in Tab. 6. For every dataset, we trained the WGANs for 2×10^5 total iterations, each of which was composed by 5 and 1 discriminator and generator updates respectively. As for the optimization, we rely on Adam [3] with a learning rate of 10^{-4}, fixing $\beta_1 = 0.5$ and $\beta_2 = 0.9$. The batch size was set to 64. The weight for gradient penalty [2] was set to 10. Inputs were normalized before being fed to the discriminator. Specifically, for MNIST we normalize each image into the range $[0, 1]$, whilst for other datasets we map inputs into the range $[-1, 1]$.

2.1. On mixing real and fake images for rehearsal.

The common practice when adopting generative replay for continual learning is to exploit a generative model to synthesize examples for prior tasks $\{1, \dots, t-1\}$, while utilizing real examples as representative of the current task t. In early experiments we followed this exact approach, but it led to sub-optimal results. Indeed, the task classifier consistently reached good discrimination capabilities during training, yielding very poor performances at test time.

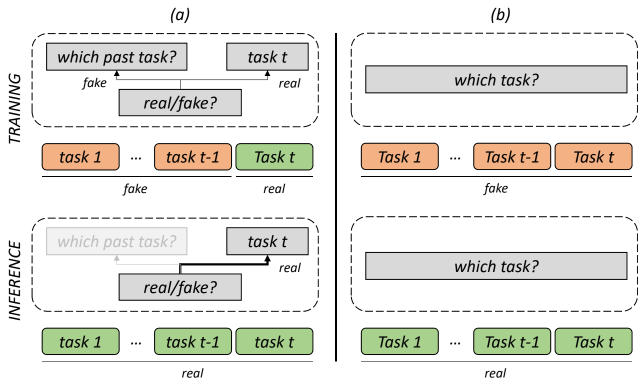

Figure 1: Illustration of (a) the degenerate behavior of the task classifier when rehearsed with a mix of real and generated examples and (b) the proposed solution. See Sec 2.1 for details.

After an in-depth analysis, we conjectured that the task classifier, while being trained on a mixture of real and fake examples, fell into the following very poor classification logic (Fig. 1). It first discriminated between the nature of the image (real/fake), learning to map real examples to task t. Only for inputs deemed as fake, a further categorization into tasks $\{1, \dots, t-1\}$ was carried out. Such a behavior, perfectly legit during training, led to terrible test performances. Indeed, during test only real examples are presented to the network, causing the task classifier to consistently label them as coming from task t.

To overcome such an issue, we remove mixing of real and fake examples during rehearsal, by presenting to the task classifier fake examples also for the task t. In the incremental learning paradigm, this only requires to shift the training of the WGAN generators from the end of a given task to its beginning.

		$C = 500$	$C = 1000$	$C = 1500$	$C = 2000$
MNIST	Full Replay	0.9861	0.9861	0.9861	0.9861
	A-GEM [1]	0.1567	0.1892	0.1937	0.2115
	iCaRL-rand [6]	0.8493	0.8455	0.8716	0.8728
	iCaRL-mean [6]	0.8140	0.8443	0.8433	0.8426
	ours	**0.9401**	**0.9594**	**0.9608**	**0.9594**
SVHN	Full Replay	0.9081	0.9081	0.9081	0.9081
	A-GEM [1]	0.5680	0.5411	0.5933	0.5704
	iCaRL-rand [6]	0.4972	0.5492	0.4788	0.5484
	iCaRL-mean [6]	0.5626	0.5469	0.5252	0.5511
	ours	**0.6745**	**0.7399**	**0.7673**	**0.8102**

Table 2: Numerical results for Fig. 4 in the main paper. Average accuracy for the episodic memory experiment, for different buffer sizes (C).

		SVHN		CIFAR-10	
		Acc.	MB	*Acc.*	MB
episodic	Em1	0.6745	1.46	0.6991	1.46
	Em2	0.7399	2.93	0.7540	2.93
	Em3	0.7673	4.39	0.7573	4.39
	Em4	0.8102	5.86	0.7746	5.86
	Em5	0.8600	32.22	0.8132	32.22
gen.	DGM [5]	0.7438	15.82	-	-
	Gm1	0.8341	33.00	0.7006	33.00

Table 3: Numerical values for the memory consumption experiment represented in Fig. 5 of the main paper.

3. Quantitative results for figures

To foster future comparisons with our work, we report in this section quantitative results that are represented in Fig. 4 and 5 of the main paper. Such quantities can be found in Tab. 2 and 3 respectively.

4. Comparison w.r.t. conditional generators

To validate the beneficial effect of the employment of generated examples for the rehearsal of task prediction only, we compare our model based on generative memory (Sec. 4.4 of the main paper) against a further baseline. To this end, we still train a WGAN-GP for each task, but instead of training *unconditional* models we train *class-conditional* ones, following the AC-GAN framework [4]. After training N conditional generators, we train the backbone model by generating labeled examples in an i.i.d fashion. We refer to this baseline as C-Gen, and report the final results in Tab. 4. The results presented for Split SVHN and Split CIFAR-10, illustrate that generative rehearsal at a task level, instead of at a class level, is beneficial in both datasets. We believe our method behaves better for two reasons. First, our model never updates classification heads guided by a loss function computed on generated examples (i.e., potentially poor in visual quality). Therefore, when the task label gets predicted correctly, the classification accuracy is comparable to the one achieved in a task-incremental setup. Moreover, given equivalent generator capacities, conditional gen-

	class conditioning	rehearsal level	SVHN	CIFAR-10
C-Gen	✓	class	0.7847	0.6384
ours	✗	task	**0.8341**	**0.7006**

Table 4: Performance of our model based on generative memory against a baseline comprising a class-conditional generator for each task (C-Gen).

erative modeling may be more complex than unconditional modeling, potentially resulting in higher degradation of generated examples.

5. Confidence of task-incremental results

To validate the gap between our model's performance with respect to HAT (Tab. 1 in the main paper), we report the confidence of such experiment by repeating it 5 times with different random seeds. Results in Tab. 5 show that the margin between our proposal and HAT is slight, yet consistent.

	MNIST	SVHN	CIFAR-10
HAT	$0.997 \pm 4.00\text{e}{-4}$	$0.964 \pm 1.72\text{e}{-3}$	$0.964 \pm 1.20\text{e}{-3}$
our	$0.998 \pm 4.89\text{e}{-4}$	$0.974 \pm 4.00\text{e}{-4}$	$0.966 \pm 1.67\text{e}{-3}$

Table 5: Task-IL results averaged across 5 runs.

References

[1] Arslan Chaudhry, MarcAurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. Efficient lifelong learning with a-gem. In *International Conference on Learning Representations*, 2019. 2

[2] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Neural Information Processing Systems*, 2017. 1

[3] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2014. 1

[4] Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier gans. In *International Conference on Machine Learning*, 2017. 2

[5] Oleksiy Ostapenko, Mihai Puscas, Tassilo Klein, Patrick Jahnichen, and Moin Nabi. Learning to remember: A synaptic plasticity driven framework for continual learning. In *IEEE International Conference on Computer Vision and Pattern Recognition*, 2019. 2

[6] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *IEEE International Conference on Computer Vision and Pattern Recognition*, 2017. 2

	Generator	Discriminator
Split MNIST	Linear(128,4096)	Conv2d(1,64,ks=(5,5),s=(2, 2))
	ReLU	ReLU
	Reshape(256,4,4)	Conv2d(64,128,ks=(5,5),s=(2, 2))
	ConvTranspose2d(256,128,ks=(5,5))	ReLU
	ReLU	Conv2d(64,128,ks=(5,5),s=(2,2))
	ConvTranspose2d(128, 64, ks=(5,5))	ReLU
	ReLU	Flatten
	ConvTranspose2d(64, 1, ks=(8,8), s=(2,2))	Linear(4096,1)
	Sigmoid	
Split SVHN	Linear(128,8192)	Conv2d(3,128,ks=(3,3),s=(2,2))
	BatchNorm1d	LeakyReLU(ns=0.01)
	ReLU	Conv2d(128,256,ks=(3,3),s=(2,2))
	Reshape(512,4,4)	LeakyReLU(ns=0.01)
	ConvTranspose2d(512,256,ks=(2,2))	Conv2d(256,512,ks=(3,3),s=(2,2))
	BatchNorm2d	LeakyReLU(ns=0.01)
	ReLU	Flatten
	ConvTranspose2d(256, 128, ks=(2,2))	Linear(8192,1)
	BatchNorm2d	
	ReLU	
	ConvTranspose2d(128, 3, ks=(2,2), s=(2,2))	
	TanH	
Split CIFAR-10	Linear(128,8192)	Conv2d(3,128,ks=(3,3),s=(2,2))
	BatchNorm1d	LeakyReLU(ns=0.01)
	ReLU	Conv2d(128,256,ks=(3,3),s=(2,2))
	Reshape(512,4,4)	LeakyReLU(ns=0.01)
	ConvTranspose2d(512,256,ks=(2,2))	Conv2d(256,512,ks=(3,3),s=(2,2))
	BatchNorm2d	LeakyReLU(ns=0.01)
	ReLU	Flatten
	ConvTranspose2d(256, 128, ks=(2,2))	Linear(8192,1)
	BatchNorm2d	
	ReLU	
	ConvTranspose2d(128, 3, ks=(2,2), s=(2,2))	
	TanH	
Imagenet-50	Linear(128,8192)	Conv2d(3,128,ks=(3,3),s=(2,2))
	BatchNorm1d	LeakyReLU(ns=0.01)
	ReLU	Conv2d(128,256,ks=(3,3),s=(2,2))
	Reshape(512,4,4)	LeakyReLU(ns=0.01)
	ConvTranspose2d(512,256,ks=(2,2))	Conv2d(256,512,ks=(3,3),s=(2,2))
	BatchNorm2d	LeakyReLU(ns=0.01)
	ReLU	Flatten
	ConvTranspose2d(256, 128, ks=(2,2))	Linear(8192,1)
	BatchNorm2d	
	ReLU	
	ConvTranspose2d(128, 3, ks=(2,2), s=(2,2))	
	TanH	

Table 6: Architecture of the WGAN employed for the generative experiment. In the table, *ks* indicates kernel sizes, *s* identifies strides, and ns refers to the negative slope of Leaky ReLU activations.