CL-Gym: Full-Featured PyTorch Library for Continual Learning

Seyed Iman Mirzadeh
Washington State University
seyediman.mirzadeh@wsu.edu

Hassan Ghasemzadeh
Washington State University
hassan.ghasemzadeh@wsu.edu

Abstract

Continual learning (CL) has become one of the most active research venues within the artificial intelligence community in recent years. Given the significant amount of attention paid to continual learning, the need for a library that facilitates both research and development in this field is more visible than ever. However, CL algorithms' codes are currently scattered over isolated repositories written with different frameworks, making it difficult for researchers and practitioners to work with various CL algorithms and benchmarks using the same interface. In this paper, we introduce CL-Gym *, a full-featured continual learning library that overcomes this challenge and accelerates the research and development. In addition to the necessary infrastructure for running end-to-end continual learning experiments,* CL-Gym *includes benchmarks for various CL scenarios and several state-of-the-art CL algorithms. In this paper, we present the architecture, design philosophies, and technical details behind* CL-Gym [1].

1. Overview

We first provide a high-level overview of the *CL-Gym* architecture and an illustrative example of how users can work with the library. *CL-Gym* includes three main components: `Benchmarks`, `Algorithms`, and `Trainer`. In addition, *CL-Gym* contains other minor components such as `Backbones` or `Callbacks`. We briefly introduce these components here and postpone the detailed discussion of each component to Section 3.

`Benchmarks` are responsible for implementing continual learning scenarios such as New Instance (NI), New Class (NC), or both. Each benchmark implements the necessary code for loading task-specific training data. Given our experience in CL research, we have also delegated the task of loading the episodic memory and loading joint/multitask training data to the benchmarks. Note that the `Algorithm` still has access to `Benchmark` and can create a separate episodic memory.

[1] https://github.com/imirzadeh/CL-Gym

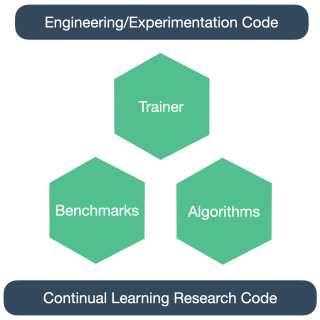

Figure 1. Main Components of *CL-Gym*

The `Algorithm` component is responsible for the majority of the research code. Each algorithm implements necessary codes for different functions of a continual learning method: training on each task, possibly updating episodic memory (e.g., ER-Ring [3]), regularization (e.g., EWC [7]), and gradient manipulation (e.g., A-GEM [2], OGD [5]). We note that the algorithm component in *CL-Gym* is not responsible for implementing the neural network model, and this responsibility is delegated to the `Backbone` component. Finally, the `Trainer` component handles non-essential research codes and engineering codes such as experiment management, collecting metrics, logging, training on different devices, and etc.

We illustrate how different components of *CL-Gym* interact with each other using the following code:

```
import cl_gym as cl

# Rotated MNIST benchmark with 5 tasks
benchmark = cl.benchmarks.RotatedMNIST(
            num_tasks=5,
            per_task_rotation_degrees=10,
            per_task_memory_examples=10)

# backbone of the algorithm: a 2-Layer MLP model
backbone = cl.backbones.MLP2Layers()

# algorithm: handles research code
#e.g., EWC, AGEM, OGD, MCSGD, ER
algorithm = cl.algorithms.EWC(backbone, benchmark)

# trainer: handles engineering code
trainer = cl.trainer.ContinualTrainer(algorithm)
trainer.fit()
```

2. Design Philosophy

CL-Gym is inspired by many popular libraries such as PyTorch Lightning [4] and PyTorch Geometric [6]. More specifically, the architecture of *CL-Gym* is designed with the following principles in mind:

1. Research code should be completely separated from the engineering code: For instance, distributed training or experiment management should not happen in the same place as the algorithmic logic.

2. Components should be open for extension but closed for modification: For example, it should be easy to build upon many algorithms or benchmarks, but the internal abstractions of each component should remain the same.

3. Many scenario-specific components are better than one general-purpose component: For instance, instead of having a general interface for all regularization algorithms, each algorithm should have its own interface.

We believe these principles help with both the development and usability of the *CL-Gym* library.

3. CL-Gym Components

In this section, we discuss each of the *CL-Gym* components in detail. In Section 4, we will present our roadmap for each component in future releases.

3.1. Benchmarks

The `Benchmark` component includes several standard continual learning benchmarks from various domains. Each benchmark can have a specific number of tasks, where for each task, a dataset is required. This abstraction allows for creating benchmarks in New Instance (NI) scenarios where for each task, a new dataset is required (e.g., rotated MNIST), New Class (NC) scenarios where the dataset for each task can be a subset of a larger dataset (e.g., split MNIST). We note that the same abstraction works for task-agnostic scenarios, and the task-specific details (e.g., task-identifiers, heads) can be hidden from the algorithm.

Moreover, each benchmark should implement the following methods:

- `prepare_datasets`: Where the customized logic for loading the datasets for each task is implemented.
- `load(task)`: for providing training and validation loaders to the algorithm.
- `load_memory(task)`: For providing episodic memory loaders. This method is optional if the benchmark is not supposed to support episodic memory (i.e., by setting memory size to 0).
- `load_joint(task)`: For providing data loaders for multitask/joint training, where for task t, the all datasets for tasks 1 to t will be loaded together. This method is optional if the benchmark is not initialized to support joint training (i.e., by setting the joint training budget to 0).
- `load_memory_joint(task)`: This method allows loading the episodic memory for all previous tasks and not for a single task. This method will be useful for rehearsal methods in NI scenarios. This method is also optional if the benchmark is initialized without a memory size.

The current implemented benchmarks are described in Table 1.

Benchmark	Type	Scenario	Details
Rotated MNIST	Vision	NI	Rotation of MNIST digits
Permuted MNIST	Vision	NI	Shuffled pixels of MNIST
Split MNIST	Vision	NC	Includes 5 task, each introduces 2 new digits.
Split CIFAR-10	Vision	NC	Includes 5 task, each introduces 2 new classes.
Split CIFAR-100	Vision	NC	Includes 20 tasks, each introduces 5 new classes.
Toy Classification	Toy - 2D	NI	Includes 2 or 4 tasks, each a binary classification.
Toy Regression	Toy - 1D	NC	Includes between 2 to 5 tasks, each a 1D regression problem
PAMAP2	Time Series	NI/NC	Includes 9 tasks, each a human activity recognition using wearable sensors.

Table 1. Supported benchmarks in *CL-Gym*

In addition to classical vision benchmarks, *CL-Gym* supports two toy datasets for classification and regression, which facilities the research on CL by allowing working in a low-dimensional regime. We have found empirically that even over-parameterized neural networks suffer from catastrophic forgetting in low-data regimes. Moreover, to bring more diverse benchmarks to the CL research, we include the PAMAP2 dataset [14], a time series dataset for human activity classification using wearable sensors.

Finally, we note that finding valuable benchmarks in continual learning is as challenging as overcome algorithmic challenges such as catastrophic forgetting. We are currently working on providing helpful benchmarks for CL research that are also computationally cheap. In Section 4, we will discuss our future plans for providing more diverse benchmarks.

3.2. Backbones

The `Backbone` component in *CL-Gym* refers to neural network models that include additional features required in CL settings. `Backbone` inherits from the PyTorch `nn` module and supports all the features this module provides.

Algorithm 1: Behavior of an algorithm on a benchmark.

```
Input: backbone, benchmark, params
initialize;
/* train on all tasks                                                    */
for task ∈ [1, benchmark.num_tasks] do
    /* load training data and optimizers                                 */
    optimizer = prepare_optimizer(task);
    batches = prepare_train_data(task);
    for epoch ∈ params.num_epochs do
        for batch ∈ batches do
            /* training step                                             */
            train_step(backbone, batch, optimizer);
            /* task-end hook (optional) : additional update              */
            /* e.g.: EWC regularization, or A-GEM gradient manipulation  */
            train_step_end();
        end
        /* epoch-end hook (optional): additional works after batch       */
        /* e.g., updating episodic memory                                */
        train_epoch_end();
    end
    /* task-end hook (optional): additional work after each task         */
    /* e.g., updating A-GEM episodic memory, EWC consolidation, or OGD orthogonal basis update */
    task_end();
end
```

The most notable feature of `Backbone` would be supporting multiple classification heads in required scenarios (e.g., NC). The algorithm module will have complete access to the backbone model and can manipulate gradients, or freeze layers.

Currently, *CL-Gym* supports the MLP architecture with two hidden layers, ResNet18 with three times fewer feature maps used in several research papers [2, 3, 11, 10], and a simple one-dimensional CNN network for time-series benchmarks. The choice of backbones is due to the number of times they have used in recent research papers.

3.3. Algorithms

The `Algorithm` component is responsible for most of the research code of in *CL-Gym* . We start this section by providing the abstraction of an algorithm in *CL-Gym* , followed by the description of methods each algorithm should implement.

Algorithm 1 represents an abstract behavior of every algorithm implemented in *CL-Gym* . This abstraction allows for implementing a variety of algorithms we introduce in the next section. The main methods each algorithm can implement are:

- `prepare_optimizer`: Creating an optimizer for each task. This allows delegating the choice of optimizer and its parameters, which are part of the research code, to the algorithm component.
- `prepare_train_data`: Loading training data using the `benchmark` interface. This allows an algorithm to control the training data with possible modifications.
- `train_step(backbone, batch, optimizer)`: The most important method an algorithm should implement. It will include algorithm-specific logic for the main optimization step for each batch. For instance, EWC applies an additional loss in this step, while A-GEM, OGD, and MC-SGD manipulate gradients.

In addition, algorithms have the option of using customized hooks at different stages of training, including:

- `train_step_end`: For adding specific behavior after training on each batch.
- `train_epoch_end`: For adding specific behavior at the end of each epoch.
- `train_task_end`: Allows adding custom behavior at the of each task. Methods that use episodic memory can update their memory in this stage, or methods such as EWC can consolidate parameters in this step. Also, methods such as OGD can update their orthogonal basis in this step.

Algorithm	Details
Elastic Weight Consolidation (EWC) [7]	Regularization
Averaged Gradient Episodic Memory (AGEM) [2]	Episodic Memory
Experience Replay RingBuffer (ER-Ring) [3]	Episodic Memory
Orthogonal Gradient Descent (OGD) [5]	Gradient Memory
Stable SGD (SSGD) [11]	Optimal training regime
Mode Connectivity SGD (MCSGD) [10]	Regularization & Memory

Table 2. Supported algorithms in *CL-Gym* , in addition to baselines such as naive finetuning or joint training (i.e., multitask)

3.3.1 Supported Algorithms

Table 2 shows the supported continual learning algorithms in *CL-Gym* that includes a diverse family of methods (e.g., regularization, rehearsal). The criteria for implementing these algorithms are their performance and their usage, among other continual learning papers. We aim to extend this component in future releases.

3.4. Trainer

The `Trainer` component is responsible for most of the non-essential research code and engineering code. These responsibilities include managing a continual learning experiment and working with the `Algorithm` component by executing Algorithm 1.

Moreover, to accomplish its responsibility, the `Trainer` will implement a timeline for the continual learning experience, as illustrated in Figure 2. This includes breaking the learning experience into smaller intervals such as `training_step`, `training_epoch`, and `training_task`. At the start/end of each interval, the trainer will emit events that allow adding customized behavior to the trainer on the fly. For instance, at the end of `training_task`, the trainer will call the `on_after_training_task()` method which can be used for metric collection.

We note that `Trainer` component in *CL-Gym* is heavily influenced by the `Trainer` component in *PyTorch Lightning* [4] for three reasons. First, it allows separating the research code from the engineering code as explained in Section 2. Secondly, this logic is known to both researchers and engineers in the machine learning field because of the popularity of *PyTorch Lightning* and makes the usage. Finally, this implementation makes executing our future plans for integrating *CL-Gym* with *PyTorch Lightning* easier, which leads to many extra features such as distributed training, efficient deployment, and experiment management.

3.5. Utility Components

In addition to the three previous main components we discussed, *CL-Gym* includes two minor components that are helpful in continual learning experiments. These components include `Callback` for customized behavior to the `Trainer`, and `Metric` for calculation metrics (e.g., forward-transfer, backward-transfer).

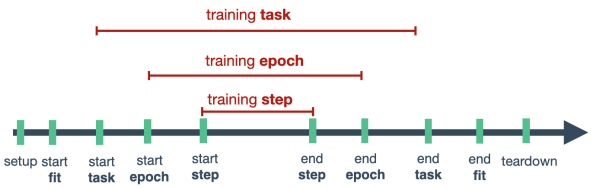

Figure 2. Trainer Timeline

3.5.1 Callbacks

A `Callback` can optionally support any of the events in Figure 2. For instance, a simple callback for collecting metrics, can implement `on_after_training_task(trainer)` and log the validation metrics to an experiment manager. Moreover, since callbacks have access to the `Trainer` object, they can add customized logic to the `Trainer`. For example, in task-agnostic scenarios, a custom `Callback` can evaluate the `Algorithm` at any given point.

3.5.2 Metrics

To evaluate the performance a continual learning algorithm, several metrics can be used. The most notable examples are forward-transfer and backward-transfer that work with the validation accuracy of an algorithm.

The `Metric` component in *CL-Gym* is responsible for collecting metrics of an `Algorithm`. The current `Metric` component, can calculate two common metrics used repeatedly in CL research [2, 3, 11, 10]:

- **Average Accuracy**: defined as

$$\frac{1}{T}\sum_{j=1}^{T} a_{T,j} \tag{1}$$

  Where T represents number of tasks, and $a_{i,j}$ denotes the test accuracy on task j after the algorithm has finished learning task i.

- **Average Maximum Forgetting**: defined as

$$\frac{1}{T-1}\sum_{j=1}^{T-1} \max_{l\in\{1,\dots,T-1\}} (a_{l,j} - a_{T,j}) \tag{2}$$

  Where T represents the number of tasks, and $a_{i,j}$ denotes the test accuracy on task j after the algorithm has finished learning task i. The average forgetting shows the decrease in performance for each of the tasks between their peak accuracy and their accuracy after the learning experience is finished.

4. Release Roadmap

In this section, we introduce the future release roadmap for *CL-Gym*. However, we note that these plans are subject to change depending on the feedback we receive from the community.

4.1. Integration with PyTorch Lightning

PyTorch Lightning (PL), is a popular library in the deep learning community which shares a similar philosophy with *CL-Gym* in separating research code from other non-research codes. In our current implementation of *CL-Gym* , the `Algorithm` component corresponds to `Lightning Module` of PL, and the `Trainer` module corresponds to the `Trainer` module of PL.

The integration with *PyTorch Lightning* adds additional features to *CL-Gym* , including:

- Distributed training and execution of on different devices such as CPUs, GPUs, and TPUs. The distributed training of algorithms will be delegated to PL.
- Experiment management by providing loggers to various experiment management services.
- Performance and bottleneck profiling.
- Access to the rich callback ecosystems for a variety of tasks such as automatic model check-pointing, gradient clipping, and layer freezing.

4.2. Additional Benchmarks

One crucial aspect of continual learning research is developing diverse and meaningful benchmarks that can help with our understanding of challenges in continual learning.

While the computationally expensive benchmarks are welcome, the core maintainers of *CL-Gym* will primarily focus on developing toy benchmarks that are easy to use for researchers. We note that even on the relatively simple Rotated MNSIT benchmark, the performance gap between the state-of-the-art CL algorithm to the joint (i.e., multitask) is quite significant and adding more difficult benchmarks may not be helpful until this gap is diminished on simpler benchmarks. Moreover, we empirically have found that even on a toy 2D classification task where the classes are linearly separable, the catastrophic forgetting happens, and the methods that perform better on MNIST and CIFAR benchmarks also perform better on this toy benchmark.

To this end, we aim to continue improving our toy benchmark submodule of the `Benchmark` component in future releases.

4.3. Additional Utility Components

4.3.1 Callbacks

After the integration with *PyTorch Lightning*, the *CL-Gym* callbacks will be modified to have the same interface as PL callback. This allows using PL callbacks in *CL-Gym* and vice versa.

4.4. Hyper-parameter Optimization

Recent research on continual learning has shown that the important role of the training regime on continual learning performance [9, 11]. Mirzadeh *et al*. [11] showed that a naive finetuning SGD method could outperform several state-of-the-art algorithms such as A-GEM with the stable training regime.

Because of the significance of the training regime, parameter optimization can play a crucial role in a CL algorithm's performance. To this end, we are planning to include a new component for hyper-parameter tuning with abstract implementations that allows integration with AutoML libraries such as Optuna [1].

4.5. CL-Gym Leaderboard

Finally, we are planning on providing a leaderboard for classical CL benchmarks using the *CL-Gym* algorithms. The leader board will include tuned algorithms in various scenarios (e.g., different number of tasks, different memory size). In addition, the experiments will be stored online with the links for the community to reproduce the results. We believe adding the leaderboard to *CL-Gym* project will help with reproducibility in continual learning research.

5. Comparison with Other Libraries

Recently, several other libraries have been released that facilitate the research on continual learning, which shows the growth of the continual learning research and the need for continual learning libraries. In this section, we compare *CL-Gym* with these libraries.

Generally, each library approaches the continual learning research differently. While *Sequoia* [12] aims to capture supervised, reinforcement, and self-supervised continual learning using the same abstraction (i.e., a hierarchy tree), *Avalanche* [8] and *CL-Gym* mostly focus on the supervised continual learning. As a result, the code base of *Sequoia* is more complex.

Sequoia

The current implementation of *Sequoia* focuses on reinforcement learning methods for continual learning, and while it can support supervised learning methods (e.g., A-GEM, MC-SGD), those methods have not been implemented yet. Moreover, *Sequoia* does mix engineering and

research code in a single place, while as explained in Section 2, we believe this increases the complexity of the library.

Finally, to the best of our knowledge, the current version of *Sequoia* does not support hooks and callbacks except when each task ends. We believe this makes implementations of several supervised-learning algorithms (e.g., OGD, MC-SGD) very difficult.

However, we note that these are architectural decisions and comes to the preferences of different users.

Avalanche

Avalanche and *CL-Gym* are very similar to each other. They both share the benchmark, backbone, and training components. Moreover, both have very similar callback and hook mechanisms.

However, evaluation and logging in *Avalanche* are implemented as main components. In contrast, in *CL-Gym* , evaluation/logging is done by updating metrics/calling the experiment manager at different stages of training, using the `Trainer` callbacks. Moreover, *Avalanche* supports more computer vision benchmarks while *CL-Gym* supports toy datasets and time-series benchmarks. In addition, *Avalanche* implements more classical continual learning algorithms while *CL-Gym* implements more recent algorithms (e.g., OGD, MC-SGD, and Stable SGD).

Finally, we emphasize that different design philosophies and implementations by *Avalanche*, *Sequoia*, and *CL-Gym* yield to different use-case scenarios that might be suitable for different groups of users.

6. Conclusion

In this paper, we presented *Cl-Gym*, a full-featured PyTorch [13] library for continual learning research and development. We explored the architecture, features of several components in *Cl-Gym* such as `Algorithms`, `Benchmarks`, and `Trainer` that are helpful for using designing new CL methods, or using several established methods. Moreover, we discussed the future roadmap of *CL-Gym* for future releases.

Continual learning is one of the most active research areas within the AI community. We believe *CL-Gym* is an important contribution to continual learning research by preparing the necessary technical infrastructure for researchers and engineers.

CL-Gym will always remain an open-source library, and we will welcome the community feedback and contributions to *CL-Gym* project.

Acknowledgements

This work was supported in part by the United States National Science Foundation, under grant CNS-1750679. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations. The authors thank Mehrdad Farajtabar and Anonymous Reviewers for their valuable comments and feedback.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631, 2019.

[2] Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. Efficient lifelong learning with a-gem. In *International Conference on Learning Representations*, 2018.

[3] Arslan Chaudhry, Marcus Rohrbach, Mohamed Elhoseiny, Thalaiyasingam Ajanthan, Puneet K. Dokania, Philip H. S. Torr, and Marc'Aurelio Ranzato. On tiny episodic memories in continual learning. *arXiv preprint arXiv:1902.10486*, 2019.

[4] William Falcon and Kyunghyun Cho. A framework for contrastive self-supervised learning and designing a new approach. *arXiv preprint arXiv:2009.00104*, 2020.

[5] Mehrdad Farajtabar, Navid Azizan, Alex Mott, and Ang Li. Orthogonal gradient descent for continual learning. In *International Conference on Artificial Intelligence and Statistics*, pages 3762–3773, 2020.

[6] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[7] James Kirkpatrick, Razvan Pascanu, Neil C. Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences of the United States of America*, 114(13):3521–3526, 2017.

[8] V. Lomonaco, Lorenzo Pellegrini, A. Cossu, Antonio Carta, G. Graffieti, Tyler L. Hayes, Matthias De Lange, Marc Masana, Jary Pomponi, Gido M. van de Ven, Martin Mundt, Qi She, Keiland W Cooper, Jeremy Forest, Eden Belouadah, S. Calderara, G. I. Parisi, Fabio Cuzzolin, A. Tolias, Simone Scardapane, L. Antiga, Subutai Amhad, A. Popescu, Christopher Kanan, J. Weijer, T. Tuytelaars, D. Bacciu, and D. Maltoni. Avalanche: an end-to-end library for continual learning. *ArXiv*, abs/2104.00405, 2021.

[9] Seyed Iman Mirzadeh, Mehrdad Farajtabar, and H. Ghasemzadeh. Dropout as an implicit gating mechanism for continual learning. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 945–951, 2020.

[10] Seyed Iman Mirzadeh, Mehrdad Farajtabar, Dilan Gorur, Razvan Pascanu, and Hassan Ghasemzadeh. Linear mode connectivity in multitask and continual learning. In *ICLR 2021: The Ninth International Conference on Learning Representations*, 2021.

[11] Seyed Iman Mirzadeh, Mehrdad Farajtabar, Razvan Pascanu, and Hassan Ghasemzadeh. Understanding the role of training regimes in continual learning. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

[12] Fabrice Normandin and et. al. Sequoia. https://github.com/lebrice/Sequoia. Accessed: 2021-04-10.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[14] Attila Reiss and D. Stricker. Introducing a new benchmarked dataset for activity monitoring. *2012 16th International Symposium on Wearable Computers*, pages 108–109, 2012.