Renate: A Library for Real-World Continual Learning

Martin Wistuba
Amazon Web Services
marwistu@amazon.com

Martin Ferianc
University College London*
martin.ferianc.19@ucl.ac.uk

Lukas Balles
Amazon Web Services
balleslb@amazon.com

Cédric Archambeau
Amazon Web Services
cedrica@amazon.com

Giovanni Zappella
Amazon Web Services
zappella@amazon.com

Abstract

Continual learning enables the incremental training of machine learning models on non-stationary data streams. While academic interest in the topic is high, there is little indication of the use of state-of-the-art continual learning algorithms in practical machine learning deployment. This paper presents Renate, a continual learning library designed to build real-world updating pipelines for PyTorch models. We discuss requirements for the use of continual learning algorithms in practice, from which we derive design principles for Renate. We give a high-level description of the library components and interfaces. Finally, we showcase the strengths of the library by presenting experimental results. Renate may be found at https://github.com/awslabs/renate.

1. Introduction

Continual learning (CL), also known as lifelong learning or incremental learning, refers to the incremental training of machine learning (ML) models on non-stationary data streams. That is, we want to update an already trained model with new data, which may come from a different distribution than the data the model has previously been trained on. The goal is to obtain a model that performs well on both the previous data as well as the new data. As an example, consider a computer vision model trained to classify satellite images according to the type of land use (e.g., agricultural or urban). It may have been trained on images from North America and may now be confronted with images from other regions of the world.

In terms of predictive performance, the gold standard of continual learning is so-called *joint training*. That is, to retrain the model, from scratch, on all previously seen data whenever new data becomes available. This obviously comes at high—and unnecessary—computational cost since it does not reuse any knowledge from previous training runs. Joint training also requires storing all previous data, which might be prohibited by data retention policies. An alternative approach would be to fine-tune the previous model on the newly-arriving data. However, this could lead to so-called *catastrophic forgetting*, where performance on older data deteriorates while training on new data. Continual learning methods have to strike a compromise between these two extremes. Among the most performant CL methods are rehearsal-based methods [3, 14, 16], which maintain a (modestly-sized) memory of previously-seen data. When training on new data, points from the rehearsal memory are "mixed in" to counteract forgetting.

Being able to efficiently update a machine learning model over time should be a key ingredient in the deployment of reliable ML systems. It allows the system to adapt to the dynamic nature of real-world data while avoiding costly retraining from scratch. However, the focus of published work on continual learning can arguably be described as mostly academic. Common benchmark problems are simulated by splitting existing datasets or adding artificial distribution shifts. Experiments often impose unrealistically restrictive constraints such as tiny rehearsal memories.

Likewise, existing software libraries for continual learning cater to researchers. They are explicitly and deliberately designed around running continual learning experiments end-to-end in a single session. Therefore, they have several shortcomings from a practical perspective:

- They offer no or limited support to serialize the state of the CL algorithm between model updates. This is a necessity in practice, where model updates may happen days or weeks apart.

- Buffers of rehearsal-based methods are stored in main memory, severely limiting their scalability.

*Work done at Amazon

- They offer no support for automatic hyperparameter optimization (HPO) and instead rely on manually-chosen hyperparameters, which may not generalize beyond standard benchmark problems.

- They offer no support to run training jobs in a cloud environment. In the same way, distributing load across several machines is not considered a necessity (e.g., multiple HPO trials in parallel).

In this paper, we present Renate, a library designed to build retraining pipelines for PyTorch [13] models that can be deployed in practice. Renate makes minimal assumptions about data modality, problem type, and model structure. It executes each model update in a separate session and fully serializes the algorithm state. Memory buffers are implemented to stream data from disk, which enables large rehearsal memories. Training runs can be executed either locally or in the cloud via AWS SageMaker and hyperparameter optimization is supported via Syne Tune [17].

2. Design Tenets

In this section, we describe principles that guided the development of Renate.

Target Real Applications When designing a library it is important to start from the problems that prospective users are facing. We designed the library's components and interfaces based on the needs of users building real-world ML model updating pipelines. Research-related functionality—e.g., running end-to-end experiments—can be built on top of that but in this case played a secondary role in design decisions.

Support Different Data Types and Tasks Real-world machine learning is not limited to one data type or task. While some continual learning methods are tailored to classification problems [15], many are much more broadly applicable. We aim to support all data types, problem tasks, model architectures, and metrics.

Automatic Hyperparameter Optimization Tuning hyperparameters is a major burden for machine learning practitioners. This is no different in the continual learning setting, with the added complication that the optimal set of hyperparameter configurations might change over time. We designed Renate with HPO in mind and provide that functionality via Syne Tune [17]. Hyperparameters can be retuned for each model update and we support algorithms for fast repeated HPO [19].

Reuse Whenever Possible Where appropriate, we build upon and reuse existing libraries. We use PyTorch Lightning [6] for basic model training instead of writing and maintaining or own training code. We also wrap methods from Avalanche [10], an existing continual learning library designed to make research reproducible, to reuse some of their learning strategies. For hyperparameter optimization we rely on Syne Tune [17].

Support Cloud Computing Training of large models most often happens in a cloud environment. This allows users to select different instances types with different compute units and memory sizes or even clusters of instances. Offering a simple and effective way to leverage cloud resources is key when creating a solution for real-world use cases.

3. Renate

We give a high-level overview of how Renate works in Figure 1. In each update step, Renate takes as input the new data, the current model and the Renate state of the previous update. The only exception is the very first update where a randomly initialized model and no Renate state is expected. Given this input, a continual learning algorithm will update the model. If the user enabled automated hyperparameter optimization, multiple training jobs will be run where each job evaluates a different hyperparameter configuration according to the hyperparameter optimization algorithm. The outputs of this process are the updated model, the new Renate state as well as log files.

Renate consists of four main components. Data access is defined by `RenateDataModule` and models by `RenateModule`. CL strategies are represented by the `Learner` class. The `ModelUpdater` coordinates these components and is responsible for storing and reloading the state. In the following, we will describe the workflow and components in more detail.

3.1. User Inputs

```python
def model_fn(model_state_url=None):
    if model_state_url is None:
        return MyModule(hp=1)
    return MyModule.from_state_dict(
        torch.load(model_state_url)
    )

def data_module_fn(data_path):
    return MyDataModule(data_path)
```

Code Snippet 1. The Renate configuration file provides access to model and data.

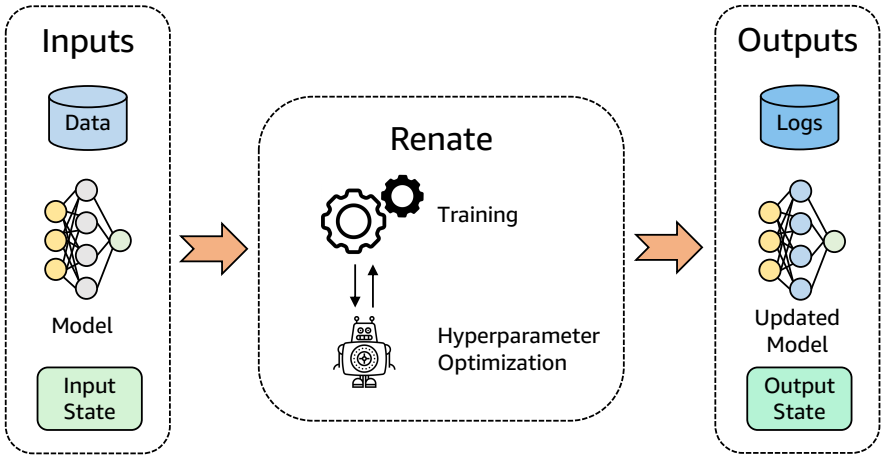

Figure 1. Renate requires as input the data, a (pretrained) model and optionally the Renate state (if Renate was used before). Renate will then update the model and optimize hyperparameters (optional). The output of this process will be the updated model and a new Renate state required for the next update step.

The user needs to provide a single Python file (Snippet 1) containing functions `model_fn` and `data_module_fn` which return an instance of `RenateModule` and a `RenateDataModule`, respectively. These instances will define the model and the data access for training. The user will need to define both these classes.

```
class MyModule(RenateModule):
    def __init__(self, hp, loss_fn):
        super().__init__(
            constructor_arguments={
                "hp": hp
            },
            loss_fn=loss_fn,
        )
        self.model = ...

    def forward(self, x):
        return self.model(x)
```

Code Snippet 2. The interface of `RenateModule` is basically identical to a PyTorch `Module`. Model hyperparameters need to registered in the base class' constructor, exemplified by `hp` here.

A `RenateModule` (Snippet 2) is a PyTorch `Module` extended by some additional logic used, among other things, for storing information to recreate an instance for future update steps. The user will need to implement the `forward` function and indicate any information besides the model itself which is required to reload the model (the constructor parameters, e.g. hyperparameters).

```
class MyDataModule(RenateDataModule):
    def prepare_data(self):
        self.download(...)

    def setup(self):
        self._train_data = ...
        self._val_data = ...
        self._test_data = ...
```

Code Snippet 3. The interface of `RenateDataModule` requires to implement the logic to make the data available on the machine and make the data accessible during training.

A `RenateDataModule` (Snippet 3) provides access to the data and two functions need to be implemented. The function `prepare_data` contains all logic required to be executed only once before accessing the data. Examples for actions taken in this function are downloading the data, extracting archived data or any other data conversions. This function is required since multiple training jobs may be executed (in parallel) on the same machine (for hyperparameter optimization) but we do not want to execute this logic more than once. The second function that needs to be implemented is called `setup` and will be executed for each training job once in the beginning and will create PyTorch `Dataset` objects for training and testing. Optionally, a validation dataset can be provided.

3.2. Training

Given the inputs, the model can be updated with `run_training_job()`. The different arguments allow for various settings. Snippet 4 provides an example for using the function.

```
run_training_job(
    mode="max",
    config_space=config_space,
    metric="accuracy",
    backend="local",
    updater="ER",
    max_epochs=50,
    input_state_url=input_state_url,
    output_state_url=output_state_url,
    config_file="renate_config.py",
)
```

Code Snippet 4. The Renate configuration file provides access to model and data.

`metric` and `mode` define the target metric and whether to minimize or maximize it. The continual learning strategy will be selected via `updater`. We discuss `backend` in more detail in Section 3.4. `config_space` will define a set of different hyperparameters, see Snippet 5 for an example. The remaining arguments are self-explanatory.

This function will trigger the model update process which will reload the previous model and trains it using the new data. For the model training we rely on Lightning [6]. This training part is divided into two components. The `ModelUpdater` controls the Lightning training process and loads and saves the state required for future updates. The `Learner` is a `LightningModule` which extends it with continual learning specific hooks which, among other things, allow to update the memory buffer. In cases where only one model needs to be trained, a `Learner` implements a specific continual learning strategy. The different hooks in the class allow for changing the loss function or update and access the memory buffer.

3.3. Hyperparameter Optimization

Renate supports hyperparameter optimization (HPO) out-of-the-box for all types of hyperparameters. It can tune the hyperparameters related to learning (e.g., learning rate, momentum), the continual learning strategy, and also user-defined hyperparameters of the model. Internally, it can use standard methods such as Bayesian optimization [18] and Asynchronous Successive Halving [9] or advanced techniques like PASHA [1] to optimize a user-defined metric such as accuracy. For that purpose, different hyperparameter configurations are evaluated and the best checkpoint with respect to that metric will be returned to the user. The logs will show the progress made during HPO and a summary is saved to a .csv file. The user can choose advanced HPO algorithms which will use the HPO history to accelerate the next HPO run.

Activating HPO in Renate is straight-forward. Instead of setting a fixed value for a hyperparameter, a space can be

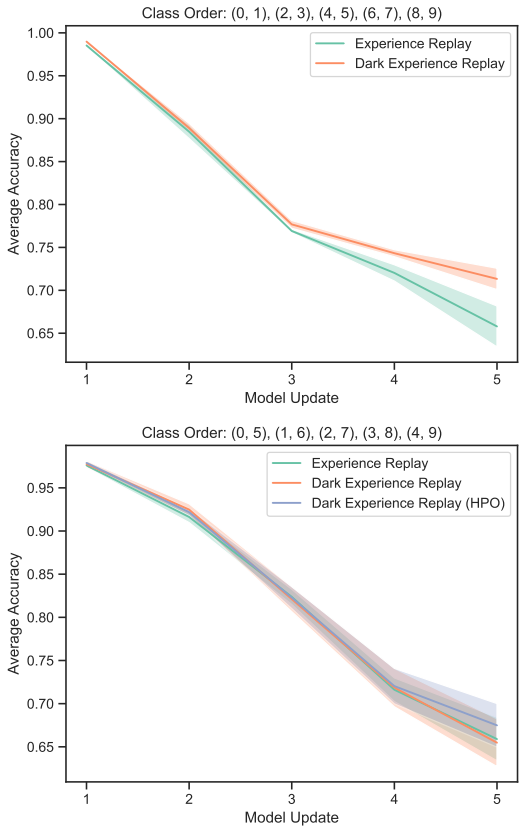

Figure 2. Selecting the right hyperparameters is not a trivial task. Top: Reproducing the results by [3]. Bottom: Changing the class order without changing anything else (including hyperparameters). DER no longer improves over the baseline. HPO alleviates this problem.

defined as seen in the following example:

```
config_space = {
  "learning_rate": loguniform(0.01, 1),
  "optimizer": choice(["SGD", "Adam"]),
  "weight_decay": 0.001,
}
```

Code Snippet 5. An example configuration. Settings with ranges are automatically optimized by HPO.

Additionally, the `max_time` argument for `run_training_job()` will define the HPO budget.

3.4. Cloud Computing

Moving the execution of a model update from the local machine to the cloud requires changing only one switch. By setting the argument `backend` of the `run_training_job()` function from "local" to "sagemaker", Renate will use AWS SageMaker. Renate currently

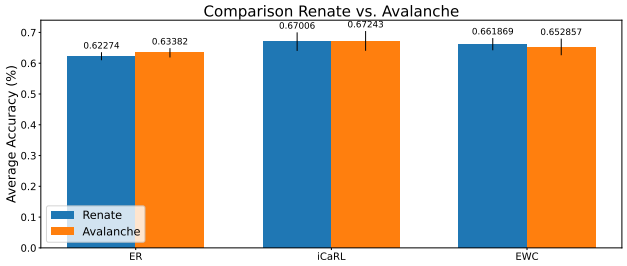

Figure 3. Comparing Avalanche algorithms by running with Avalanche or within Renate.

only supports AWS SageMaker but this is not a hard limitation. For other services the local backend should be used and training job submission scripts must be written by the user. Renate can also be extended to support further backends. To load or save the Renate files in Amazon's cloud storage service S3, simply provide S3 paths.

3.5. Outputs

The output of an Renate model update is the model checkpoint (can be loaded via `model_fn()`), the state of the continual learning strategy, and a log file containing metadata about all training jobs. The model checkpoint is the only part required for deploying the model. The state of the learning strategy is required for the next update since it may store necessary information such as the memory buffer. Finally, the metadata will fuel our advanced HPO strategies to reduce tuning time.

4. Running Experiments with Renate

In this section, we showcase Renate. Our goal is to illustrate the strenghts of the library in two scenarios: 1) benefits of hyperparameter optimization in the context of continual learning and 2) flexible APIs that facilitate the integration of other open source software.

4.1. Hyperparameter Optimization

Selecting the hyperparameters of continual learning algorithms, training algorithms and models is by no means trivial. Furthermore, the learning algorithms are sensitive to the choice of hyperparameter configurations as most works use different configurations for different benchmarks. This means, that they cannot be set once and then work in all cases. To illustrate this, we reproduced the experiment run in [3]. We consider the standard class-incremental scenario for CIFAR-10 and set the manually picked hyperparameter configurations by the authors to reproduce their comparison between Dark Experience Replay (DER) and Experience Replay (ER) using a buffer size of 500. As can be seen in the top plot of Figure 2, we are able to reproduce the results reported by the authors for DER. DER shows significantly better results than ER. In the bottom plot of the same figure we show the results for the same hyperparameter configurations with a small change: the class order is different. Keep in mind, in practice optimizing hyperparameters over the entire data sequence is not possible and therefore this is a legitimate change. What we notice is that DER does no longer show any benefits over the simpler and faster ER. Furthermore, we can see that we can mitigate this problem by optimizing the hyperparameters for each update step.

4.2. Avalanche Integration

Renate allows to use continual learning strategies implemented in Avalanche, a library which is actively maintained and provides many different continual learning algorithms. To demonstrate that, we first run a learning strategy with Avalanche only. Then, we run the same strategy with the same settings but using Renate instead. The outcome of this experiment is shown in Figure 3. ER and iCaRL [15] were evaluated on the class-incremental scenario described in the last section. EWC [8] was evaluated using an MLP and the Rotation-MNIST with 10 tasks. Results are repeated ten times. We see no statistically different predictive performance indicating that Renate successfully wrapped these learning strategies. This means that users of Renate can use the Avalanche algorithms and additionally have access to all additional algorithms provided by Renate. Furthermore, when using the Avalanche algorithms with Renate, the user will benefit from many Renate features, i.e., hyperparameter optimization, cloud support and full serialization.

5. Related Work

The scientific literature on continual learning is vast. We limit the discussion of related work to software libraries and tools for continual learning.

Several papers [4, 7, 11] have benchmarked CL algorithms and made implementations publicly available. Avalanche [10] is a PyTorch-based library, designed as a "shared and collaborative open-source [...] codebase for fast prototyping, training and reproducible evaluation of continual learning algorithms". Another PyTorch-based library with a similar design is Mammoth [2]. Both projects implement a variety of CL algorithms with standardized interfaces and enable users to run reproducible continual learning experiments end-to-end. Sequoia [12] is library with a broader scope, aiming to provide a "playground for research at the intersection of continual, reinforcement, and self-supervised learning." Finally, continuum [5] is a Python library which implements data loading functionality to create commonly-used continual learning scenarios.

These existing libraries are great tools for researchers who want to run end-to-end experiments to compare CL algorithms. Renate pursues an additional goal: support prac-

titioners using ML in real-world scenarios to efficiently and effectively update their models.

6. Conclusion

We have presented Renate, a continual learning library for PyTorch models. We formulated our design principles, motivated by the requirements of using continual learning methods to build real-world model updating pipelines. We reviewed the key components of the library and presented experiments showcasing its strengths. We in particular highlighted the need for hyperparameter optimization and the ease of integration of other open-source projects. Renate is under active development. In upcoming releases, we plan to enhance support for more problem types (e.g., regression, ranking), enable training on large models (e.g., using multiple GPUs) and to provide more state-of-the-art CL methods. We will also incorporate methods for the *detection* of data distribution shifts to address the question of *when* to trigger a model update.

References

[1] Ondrej Bohdal, Lukas Balles, Martin Wistuba, Beyza Ermis, Cedric Archambeau, and Giovanni Zappella. PASHA: Efficient HPO and NAS with progressive resource allocation. In *The Eleventh International Conference on Learning Representations*, 2023. 4

[2] Matteo Boschini and the Mammoth Team. Mammoth: An extendible (general) continual learning framework based on Pytorch. https://github.com/aimagelab/mammoth. 5

[3] Pietro Buzzega, Matteo Boschini, Angelo Porrello, Davide Abati, and Simone Calderara. Dark experience for general continual learning: a strong, simple baseline. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. 1, 4, 5

[4] Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Aleš Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. *IEEE transactions on pattern analysis and machine intelligence*, 44(7):3366–3385, 2021. 5

[5] Arthur Douillard and Timothée Lesort. Continuum: Simple management of complex continual learning scenarios, 2021. 5

[6] William Falcon and The PyTorch Lightning team. PyTorch Lightning, 2 2023. 2, 4

[7] Yen-Chang Hsu, Yen-Cheng Liu, Anita Ramasamy, and Zsolt Kira. Re-evaluating continual learning scenarios: A categorization and case for strong baselines. *arXiv preprint arXiv:1810.12488*, 2018. 5

[8] 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. *CoRR*, abs/1612.00796, 2016. 5

[9] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In *Third Conference on Systems and Machine Learning*, 2020. 4

[10] Vincenzo Lomonaco, Lorenzo Pellegrini, Andrea Cossu, Antonio Carta, Gabriele Graffieti, Tyler L. Hayes, Matthias De Lange, Marc Masana, Jary Pomponi, Gido van de Ven, Martin Mundt, Qi She, Keiland Cooper, Jeremy Forest, Eden Belouadah, Simone Calderara, German I. Parisi, Fabio Cuzzolin, Andreas Tolias, Simone Scardapane, Luca Antiga, Subutai Amhad, Adrian Popescu, Christopher Kanan, Joost van de Weijer, Tinne Tuytelaars, Davide Bacciu, and Davide Maltoni. Avalanche: an end-to-end library for continual learning. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2nd Continual Learning in Computer Vision Workshop, 2021. 2, 5

[11] Marc Masana, Xialei Liu, Bartłomiej Twardowski, Mikel Menta, Andrew D Bagdanov, and Joost van de Weijer. Class-incremental learning: Survey and performance evaluation on image classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022. 5

[12] Fabrice Normandin, Florian Golemo, Oleksiy Ostapenko, Pau Rodríguez, Matthew D. Riemer, Julio Hurtado, Khimya Khetarpal, Dominic Zhao, Ryan Lindeborg, Timothée Lesort, Laurent Charlin, Irina Rish, and Massimo Caccia. Sequoia: A software framework to unify continual learning research. *CoRR*, abs/2108.01005, 2021. 5

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. 2

[14] Roger Ratcliff. Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions. *Psychological Review*, 97(2):285–308, 1990. 1

[15] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H. Lampert. icarl: Incremental classifier and representation learning. In *CVPR*, pages 5533–5542. IEEE Computer Society, 2017. 2, 5

[16] Anthony V. Robins. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connect. Sci.*, 7(2):123–146, 1995. 1

[17] David Salinas, Matthias Seeger, Aaron Klein, Valerio Perrone, Martin Wistuba, and Cedric Archambeau. Syne tune: A library for large scale hyperparameter tuning and reproducible research. In *First Conference on Automated Machine Learning (Main Track)*, 2022. 2

[18] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. 4

[19] Giovanni Zappella, David Salinas, and Cédric Archambeau. A resource-efficient method for repeated hpo and nas. In

ICML 2021 Workshop on Automated Learning (AutoML), 2021. 2