Continual Learning: On Machines that can Learn Continually

Official Open-Access Course @ University of Pisa, ContinualAI, AIDA

Lecture 5: Methodologies [Part 1]

Vincenzo Lomonaco
University of Pisa & ContinualAI
vincenzo.lomonaco@unipi.it

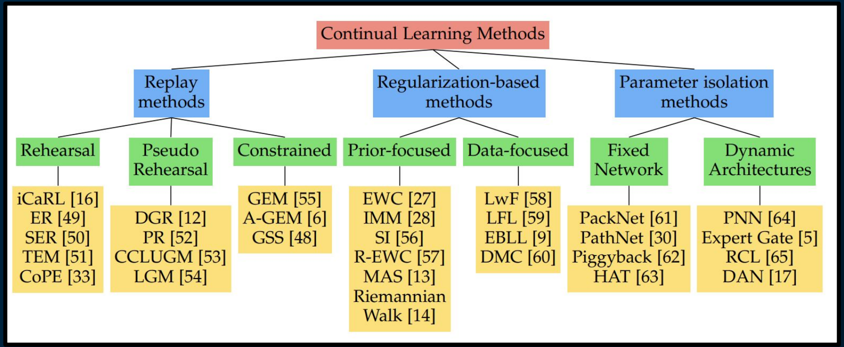
TABLE OF CONTENTS

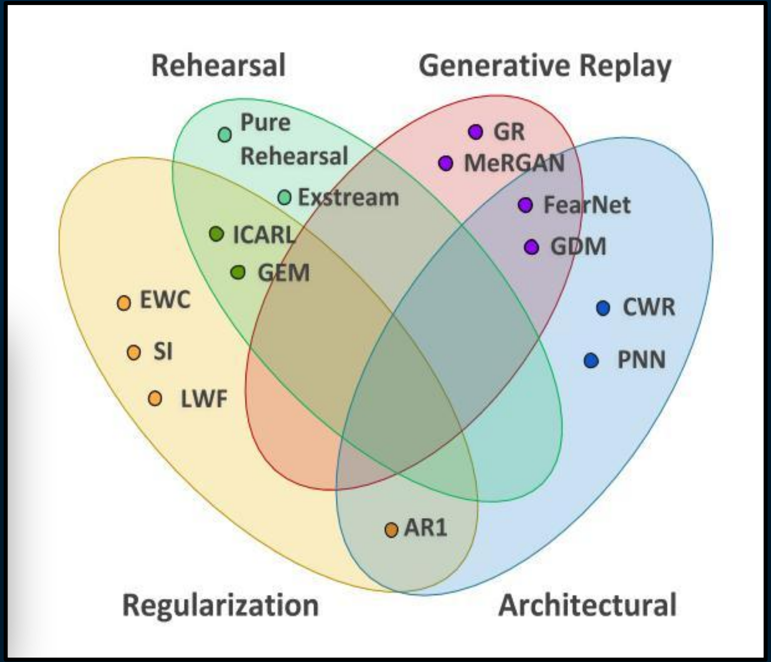
Strategy Categorization and History

Possible 4-way Fuzzy Categorization

With some twists

- **No formal definition**
- **Alternative categorizations** are possible



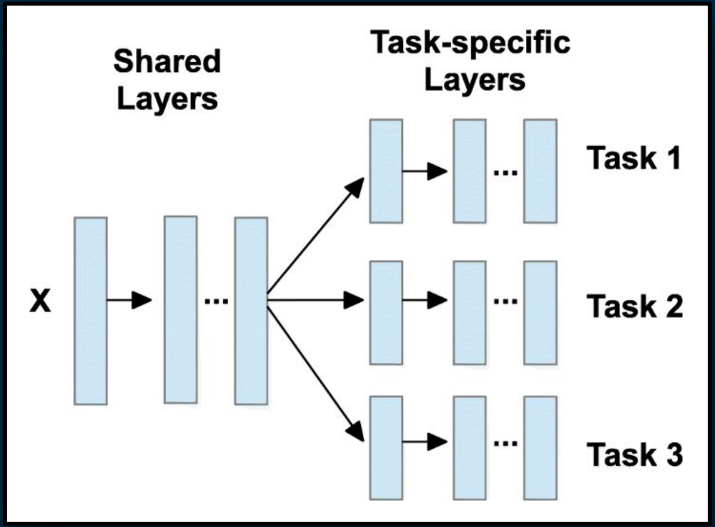


__Continual Learning for Robotics: Definition, Framework, Learning Strategies, Opportunities and Challenges__, Lesort et al. Information Fusion, 2020.
__A continual learning survey: Defying forgetting in classification tasks__. De Lange et al, TPAMI 2021.

Continual Learning Baselines

Common Baselines / Control Algorithms

- **Naive** / **Finetuning** (just continuing backprop)
- **JointTraining** / **Offline** (pure Multi-task learning): The best you can do with all the data starting from scratch
- **Ensemble**: one model for each experience
- **Cumulative**: for every experience, accumulate all data and re-train from scratch.

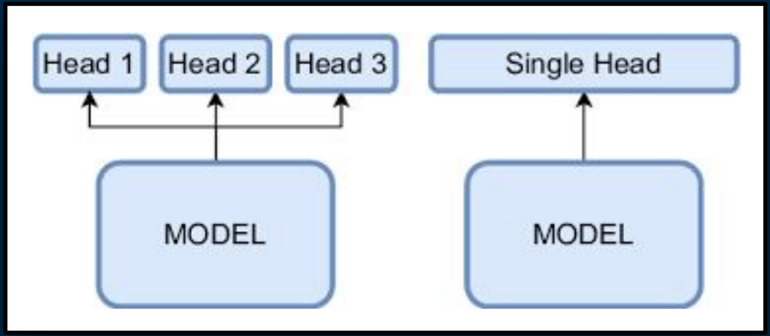


A brief review on multi-task learning. Thung et al, 2018.

Fundamental Design Choices

Strategic Choices

- Start **from scratch** or **pre-trained**?
- What **model architecture** to use?
- Such choices may **affect the CL approach effectiveness**



Multi-Head vs Single-Head

Continual Learning for Recurrent Neural Networks: an Empirical Evaluation. Cossu et al, 2021.

Historical Trends

- Initial focus on Task Incremental (a few experiences, one for task, task labels given)
- **Simple Regularization** methods (L1 / L2, Dropout, Elastic Weights Consolidation, Synaptic Intelligence, etc.)
- **Simple Architectural** strategies (Multi-head, Copy-Weight with Reinit, Progressive Neural Networks, etc.)
- **Simple Replay Strategies** (random Replay, multi-buffer random replay, etc.)
- Current trend: more and more articulate strategies (often starting from pre-trained models), mostly **hybrid**
- Mostly **Heuristics**, not principled methods. Very **difficult to generalize** to a large set of scenarios

Effective Solutions

Good News

- Replay is a **very general** and **effective** strategy for CL

Bad News

- Replay **is approximating an i.i.d distribution**
- Can be seen as a form of **cheating**
- **Compute / memory limitations**

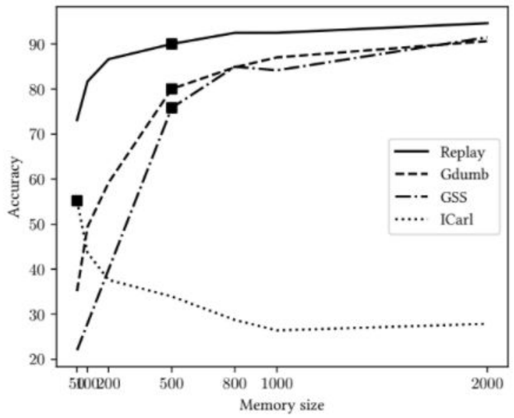


Figure 5.2: Split-MNIST memory-accuracy curve

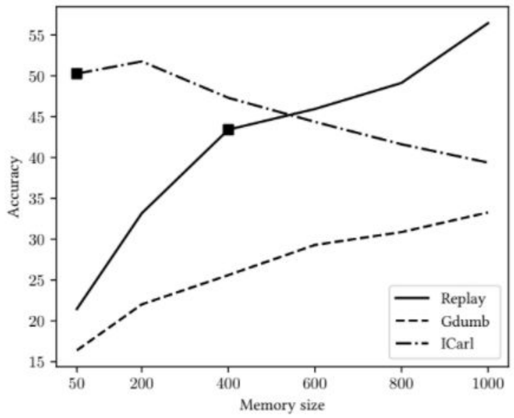


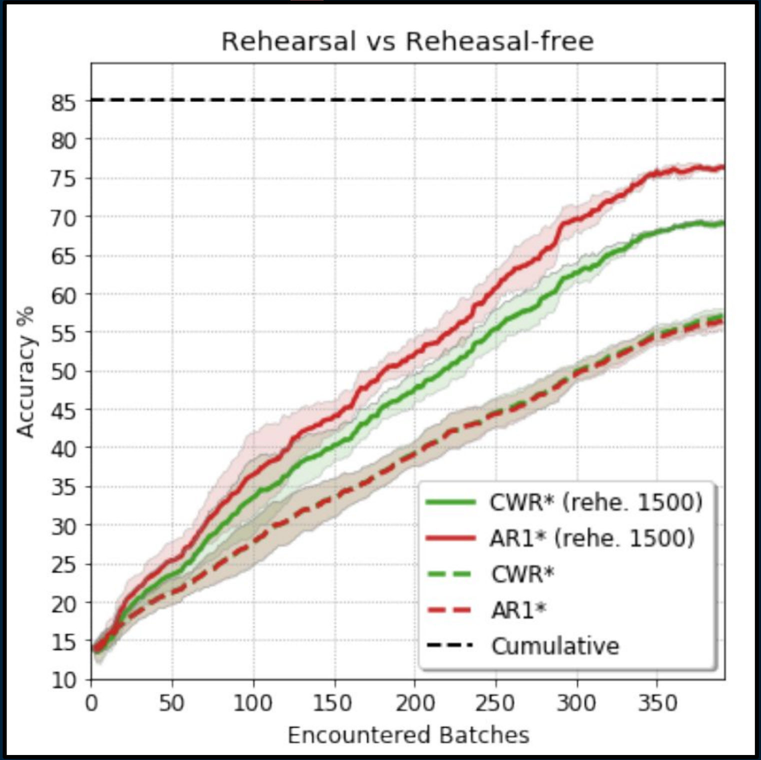
Figure 5.3: Split-CIFAR-10 memory-accuracy curve

Replay-Based Methods for Continual Learning, Gabriele Merlin, MS Thesis, University of Pisa, 2021.

Is Forgetting Solved?

Not really

- The **gap** with an offline strategy may be **still very large**
- The **accuracy improvements** with respect to the memory size is often **logarithmic**
- **Huge buffer sizes** (approximating a cumulative strategy) may be **very inefficient**
    - **Memory size** (for imagenet 50 imgs per class means about 7 GB memory)
    - **Additional forward** and **backward passes** over the same examples



Latent Replay for Real-Time Continual Learning. Pellegrini et al. IROS, 2019.

Replay Strategies

Random Replay

A basic approach

- **Sample randomly** from the current experience data
- **Fill your fixed Random Memory** (RM)
- **Replace examples randomly** to maintain an approximate equal number of examples for experience

Algorithm 1 Pseudocode explaining how the external memory RM is populated across the training batches. Note that the amount h of patterns to add progressively decreases to maintain a nearly balanced contribution from the different training batches, but no constraints are enforced to achieve a class-balancing.

$RM = \varnothing$

RM_{size} = number of patterns to be stored in RM

for each training batch B_i:

train the model on shuffled $B_i \cup RM$

$h = \frac{RM_{size}}{i}$

R_{add} = random sampling h patterns from B_i

$R_{replace} = \begin{cases} \varnothing & \text{if } i == 1 \\ \text{random sample } h \text{ patterns from } RM & \text{otherwise} \end{cases}$

$RM = (RM - R_{replace}) \cup R_{add}$

Latent Replay for Real-Time Continual Learning. Pellegrini et al. IROS, 2019.

Many Implementation Options

...and many implications

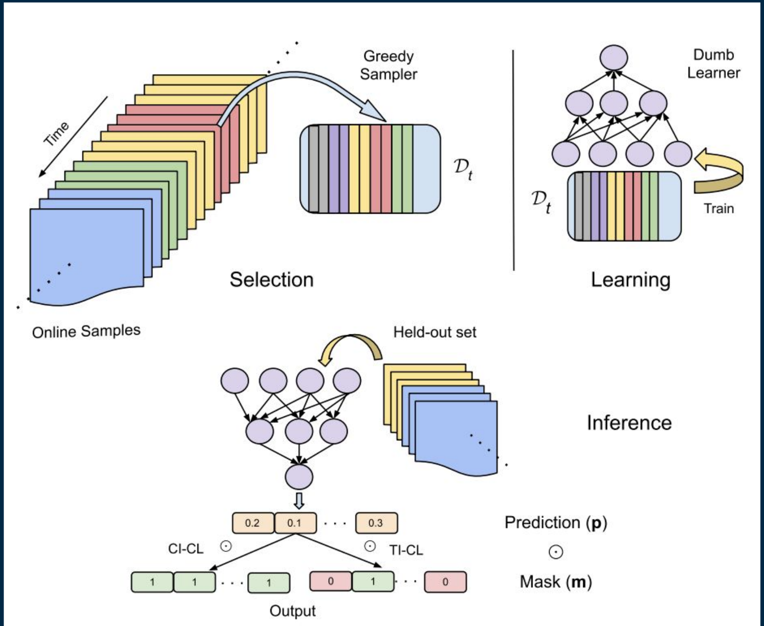
- **Fixed** or "**adaptive**" external memory?
- **Sample selection**: random or representative examples only?
- **Mini-batch sample selection**: what examples to choose from M and to use in the current mini-batch? What augmentations to use?
- **Separate buffers** per class / tasks / notable distributions?
- **Sample based on time**: different timescales? Uniform sampling in time?
- **Sample replacement**: which examples to throw away when the memory is full?
- **No clear answer** to all these questions: a coherent empirical evaluation still missing
- It really depends on the scenario / problem you are solving -> **more engineering than science**

Memory Efficient Experience Replay for Streaming Learning, Hayes et al. 2019

GDUMB: Another Control Baseline

Greedy Sampler and Dumb Learner

- Interesting paper that sparked **strong discussions** in the CL community
- Note that there's **no knowledge transfer** in this strategy (quite dumb indeed!)
- Despite its simplicity, It was shown to work better than some existing and more complex strategies, **questioning the utility of some benchmarks/metrics** in our field
- If your strategy cannot beat GDumb there's **something wrong** about your strategy or your evaluation setting

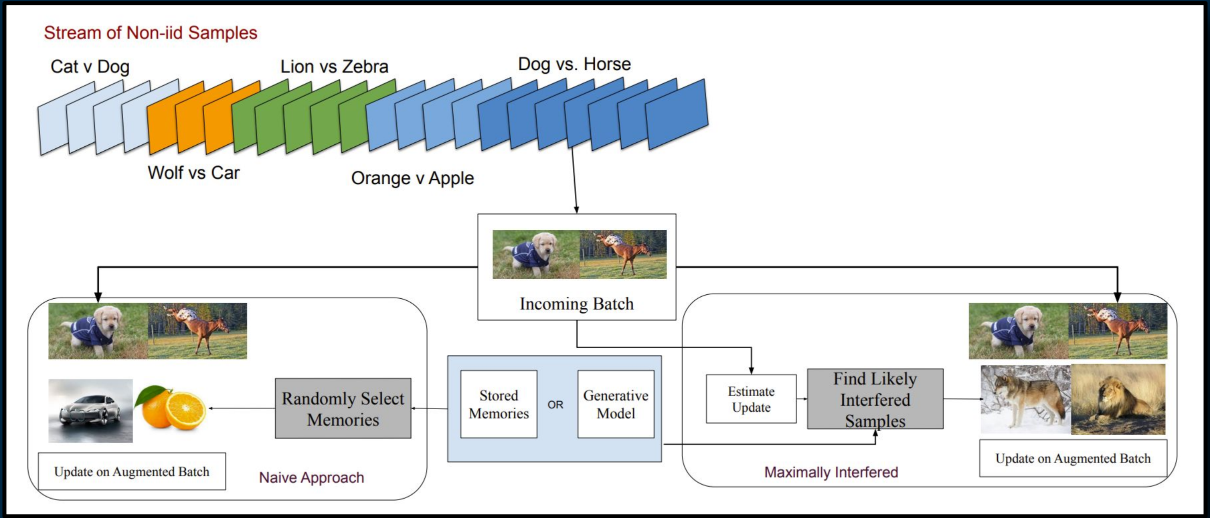


[GDumb: A Simple Approach that Questions Our Progress in Continual Learning](). Prabhu et al. ECCV, 2020.

Maximally Interfered Retrieval (MIR)

Mini-batch Sample Selection

- Select the examples that are more **negatively impacted** by the estimated weights update
- **May be quite slow in practice** w.r.t. the actual accuracy gain over random selection

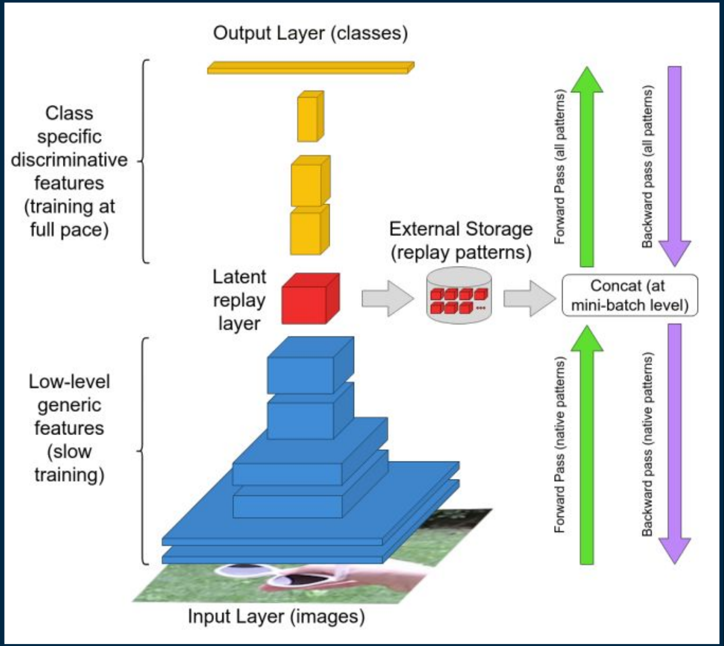


[Online Continual Learning with Maximally Interfered Retrieval](). Aljundi et al. 2019.

Latent Replay

Key Ideas

- Replay in the input space is **inefficient** and **biologically implausible**
- Why not replaying in the latent activations space?
- Good **Accuracy-Memory-Computation trade-offs** are possible

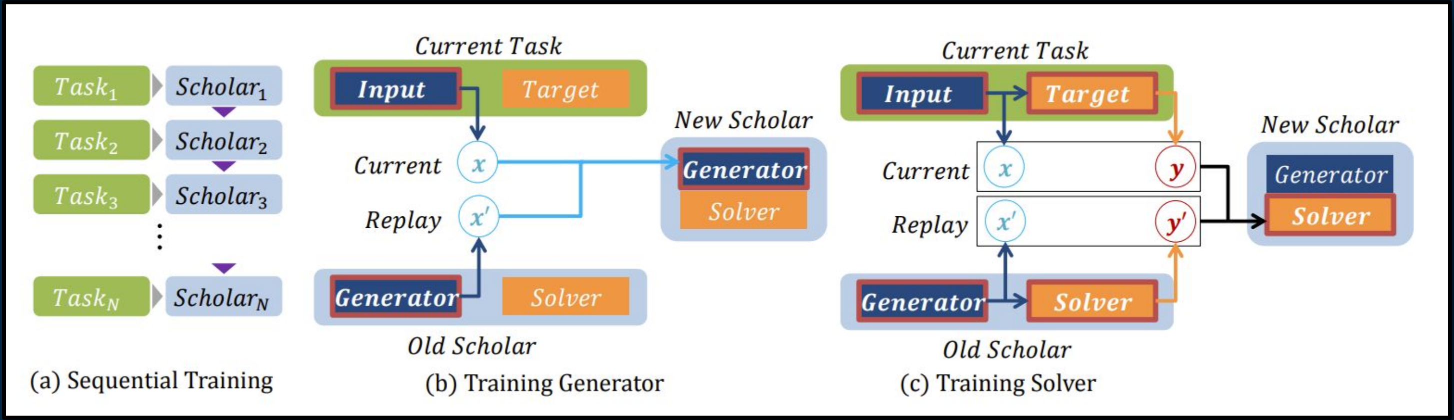


Latent Replay for Real-Time Continual Learning. Pellegrini et al. IROS, 2019.

Generative Replay

Key Ideas

- Instead of a replay memory **why not generating examples**?
- **In theory** this would be **even better than replay**: allowing for generating examples that were never seen before (a form of *dreaming* or *imagination*)
- Still **difficult** to scale on **high-dimensional data** and find good **accuracy-efficiency trade-offs**



Continual Learning with Deep Generative Replay, Shi et al, 2017.

Replay: Summary and Next Steps

- A **definitive study** of replay in deep continual learning is **still missing**
- Replay has been shown to be an **effective strategy in CL** if performance is the main objective
- Replay is **unlikely to be represent the main computational principle** for CL in biological learning systems (not a good efficiency-effectiveness trade-off)
- **Many improvements and implementation options** have been explored with different degrees of success
- Generative / latent replay constitute an interesting future direction but quite challenge at the moment due to the **limited generative models capabilities**

Avalanche

Strategies and Plugins

Training: Design

Avalanche provides popular **strategies already implemented** and ready-to-use and easy mechanisms to define **custom strategies**.

- *Many strategies are already available*
- Easy modification of the training loop to add logging and custom behavior (mostly trough **Polymorphism**)

V. Lomonaco et al. ***Avalanche: an End-to-End Library for Continual Learning***. CLVision Workshop at CVPR 2021.

How to: Strategy Initialization

```
strategy = Replay(model, optimizer,
                  criterion, mem_size)
for train_exp in scenario.train_stream:
    strategy.train(train_exp)
    strategy.eval(scenario.test_stream)
```

How to: Training & Evaluation

```python
from avalanche.benchmarks.classic import SplitMNIST

# scenario
scenario = SplitMNIST(n_experiences=5, seed=1)

# TRAINING LOOP
print('Starting experiment...')
results = []
for experience in scenario.train_stream:
    print("Start of experience: ", experience.current_experience)
    print("Current Classes: ", experience.classes_in_this_experience)

    train_res = cl_strategy.train(experience)
    print('Training completed')

    print('Computing accuracy on the whole test set')
    results.append(cl_strategy.eval(scenario.test_stream))
```

V. Lomonaco et al. ***Avalanche: an End-to-End Library for Continual Learning***. CLVision Workshop at CVPR 2021.

Training: Design

- **Strategy**: defines a CL strategy with two simple methods:
    - train and eval.
- **Plugin**: a simple interface to add custom behavior to the training and eval loops.

V. Lomonaco et al. ***Avalanche: an End-to-End Library for Continual Learning***. CLVision Workshop at CVPR 2021.

How to: Add Plugins

```
replay = ReplayPlugin(mem_size)
ewc = EWCPlugin(ewc_lambda)
strategy = BaseStrategy(
    model, optimizer,
    criterion, mem_size,
    plugins=[replay, ewc])
```

V. Lomonaco et al. ***Avalanche: an End-to-End Library for Continual Learning***. CLVision Workshop at CVPR 2021.

Training: Custom Strategies

How to write custom strategy

- **plugin**: the easiest way to customize training and define new strategies.
- **strategy**: override the loop methods directly.

Why should I use Avalanche to implement my own strategies?

- **automatic logging & metrics** evaluation.
- **you write less code**, and you can easily share it with the community.

V. Lomonaco et al. ***Avalanche: an End-to-End Library for Continual Learning***. CLVision Workshop at CVPR 2021.

BaseStrategy: Under the hood

- The **base class from which to inherit** and to specialize
- Implemented as a **series of callbacks** as a *skeleton to the plugin system*: this means you can write plugins **"by difference"** and **compose plugins**

```
train
    before_training
    before_training_exp
    adapt_train_dataset
    make_train_dataloader
    before_training_epoch
        before_training_iteration
            before_forward
            after_forward
            before_backward
            after_backward
        after_training_iteration
        before_update
        after_update
    after_training_epoch
    after_training_exp
    after_training
```

V. Lomonaco et al. ***Avalanche: an End-to-End Library for Continual Learning***. CLVision Workshop at CVPR 2021.

Custom Plugin

```
from avalanche.training.plugins import StrategyPlugin

class ReplayPlugin(StrategyPlugin):
    """ Experience replay plugin. """

    def __init__(self, mem_size=200):
        super().__init__()
        self.mem_size = mem_size
        self.ext_mem = {}  # a Dict<task_id, Dataset>
        self.rm_add = None

    def adapt_train_dataset(self, strategy, **kwargs):
        """
        Expands the current training set with datapoints from
        the external memory before training.
        """
        ...

    def after_training_exp(self, strategy, **kwargs):
        """
        After training we update the external memory with the patterns of
         the current training batch/task.
        """
        ...
```

V. Lomonaco et al. ***Avalanche: an End-to-End Library for Continual Learning***. CLVision Workshop at CVPR 2021.

Custom Strategy

```
class Cumulative(BaseStrategy):
    def __init__(*args, **kwargs):
        super().__init__(*args, **kwargs)
        self.dataset = {}  # cumulative dataset

    def adapt_train_dataset(self, **kwargs):
        """ Concatenate data from previous experiences. """
        super().adapt_train_dataset(**kwargs)
        curr_task_id = self.experience.task_label
        curr_data = self.experience.dataset
        if curr_task_id in self.dataset:
            cat_data = ConcatDataset([self.dataset[curr_task_id],
                                      curr_data])
            self.dataset[curr_task_id] = cat_data
        else:
            self.dataset[curr_task_id] = curr_data
        self.adapted_dataset = self.dataset
```

V. Lomonaco et al. ***Avalanche: an End-to-End Library for Continual Learning***. CLVision Workshop at CVPR 2021.

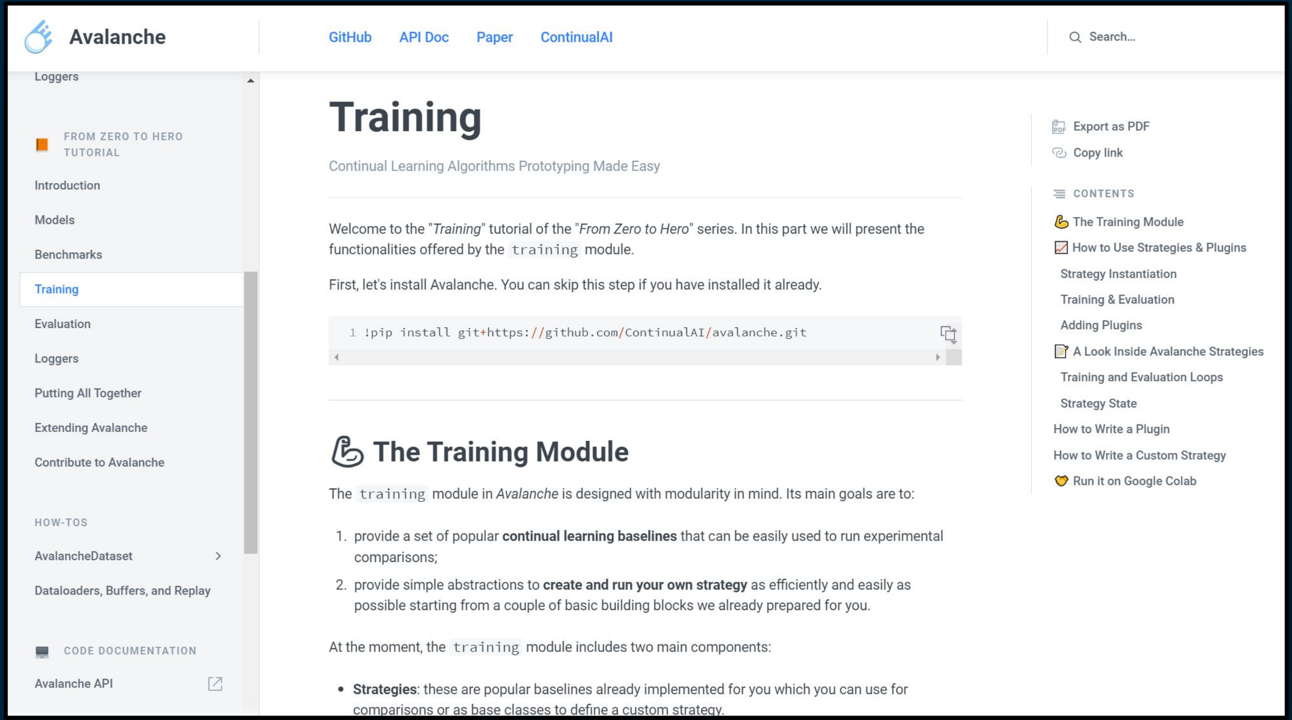
Training: What's Next?

- **More Strategies & Plugins!** (and make sure they can reproduce published results)
- **Support for Unsupervised / Reinforcement Continual Learning** (check the Avalanche ecosystem!)

V. Lomonaco et al. ***Avalanche: an End-to-End Library for Continual Learning***. CLVision Workshop at CVPR 2021.

Training in Avalanche

Demo Session!



Avalanche

GitHub API Doc Paper ContinualAI

Search...

Loggers

FROM ZERO TO HERO TUTORIAL

- Introduction
- Models
- Benchmarks
- Training
- Evaluation
- Loggers
- Putting All Together
- Extending Avalanche
- Contribute to Avalanche

HOW-TOS

- AvalancheDataset
- Dataloaders, Buffers, and Replay

CODE DOCUMENTATION

- Avalanche API

Training

Continual Learning Algorithms Prototyping Made Easy

Welcome to the "*Training*" tutorial of the "*From Zero to Hero*" series. In this part we will present the functionalities offered by the `training` module.

First, let's install Avalanche. You can skip this step if you have installed it already.

```
1 !pip install git+https://github.com/ContinualAI/avalanche.git
```

The Training Module

The `training` module in *Avalanche* is designed with modularity in mind. Its main goals are to:

1. provide a set of popular **continual learning baselines** that can be easily used to run experimental comparisons;
2. provide simple abstractions to **create and run your own strategy** as efficiently and easily as possible starting from a couple of basic building blocks we already prepared for you.

At the moment, the `training` module includes two main components:

- **Strategies**: these are popular baselines already implemented for you which you can use for comparisons or as base classes to define a custom strategy.

Export as PDF

Copy link

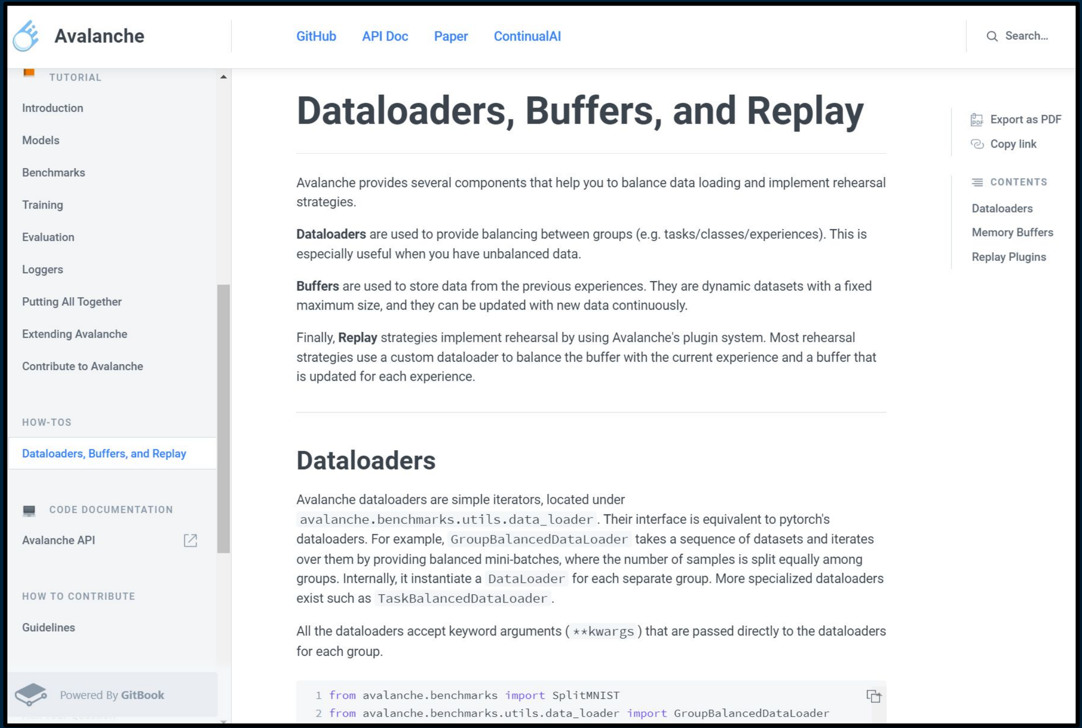
CONTENTS

https://avalanche.continualai.org/from-zero-to-hero-tutorial/04_training

Replay in Avalanche

Replay in Avalanche

Demo Session!



Avalanche

GitHub API Doc Paper ContinualAI

Search...

TUTORIAL

- Introduction
- Models
- Benchmarks
- Training
- Evaluation
- Loggers
- Putting All Together
- Extending Avalanche
- Contribute to Avalanche

HOW-TOS

- Dataloaders, Buffers, and Replay

CODE DOCUMENTATION

- Avalanche API

HOW TO CONTRIBUTE

- Guidelines

Powered By GitBook

Dataloaders, Buffers, and Replay

Export as PDF

Copy link

CONTENTS

- Dataloaders
- Memory Buffers
- Replay Plugins

Avalanche provides several components that help you to balance data loading and implement rehearsal strategies.

Dataloaders are used to provide balancing between groups (e.g. tasks/classes/experiences). This is especially useful when you have unbalanced data.

Buffers are used to store data from the previous experiences. They are dynamic datasets with a fixed maximum size, and they can be updated with new data continuously.

Finally, **Replay** strategies implement rehearsal by using Avalanche's plugin system. Most rehearsal strategies use a custom dataloader to balance the buffer with the current experience and a buffer that is updated for each experience.

Dataloaders

Avalanche dataloaders are simple iterators, located under `avalanche.benchmarks.utils.data_loader`. Their interface is equivalent to pytorch's dataloaders. For example, `GroupBalancedDataLoader` takes a sequence of datasets and iterates over them by providing balanced mini-batches, where the number of samples is split equally among groups. Internally, it instantiate a `DataLoader` for each separate group. More specialized dataloaders exist such as `TaskBalancedDataLoader`.

All the dataloaders accept keyword arguments (`**kwargs`) that are passed directly to the dataloaders for each group.

```
from avalanche.benchmarks import SplitMNIST
from avalanche.benchmarks.utils.data_loader import GroupBalancedDataLoader
```

https://avalanche.continualai.org/how-tos/dataloading_buffers_replay

Next:

Methodologies [Part 2]

Do you have any questions?

vincenzo.lomonaco@unipi.it
vincenzolomonaco.com
University of Pisa

THANKS