Avalanche

An End-to-End Library for Continual Learning

avalanche.continualai.org

powered *by*

ContinualAI

Antonio Carta, Andrea Cossu, Lorenzo Pellegrini, Gabriele Graffieti, Hamed Hemati, Vincenzo Lomonaco
and many more contributors…

About Me

- Assistant Professor @ University of Pisa
- Lead Maintainer of Avalanche @ ContinualAI
- Researcher on Continual Learning

PAILab

Avalanche

powered by

ContinualAI

https://www.continualai.org/

https://avalanche.continualai.org/

Plan for Today

- What do you need for Continual Learning?
- Avalanche API
- Example notebooks

What is Avalanche

Avalanche is a Continual Learning Library based on PyTorch

- **ContinualAI** collaborative and community-driven open-source (MIT licensed)
- **fast prototyping** and high-level API
- **reproducibility**: https://github.com/ContinualAI/continual-learning-baselines
- **modular**: you can use only a subset of Avalanche (benchmarks, models, regularization methods)
- a **consistent and general** nomenclature that covers many CL settings

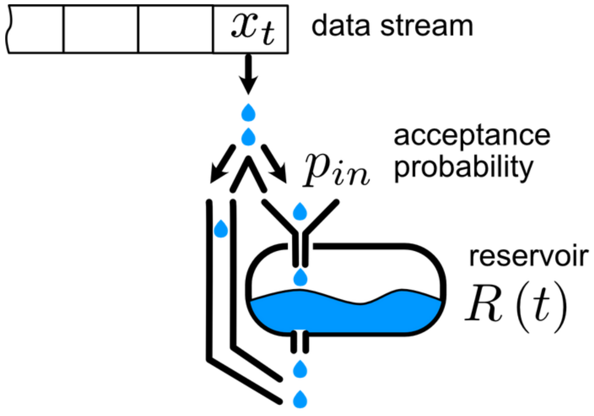
People and Organization

- **maintainers**: Vincenzo Lomonaco, Lorenzo Pellegrini, Andrea Cossu, Antonio Carta, Hamed Hemati
- many **external contributors** (50+)
- **regularly used by the community** to create new benchmarks, teaching resources or CL challenges:
  - CL Course: https://course.continualai.org/
  - CLVISION challenge: https://github.com/ContinualAI/clvision-challenge-2022
  - Endless CL Simulator: https://arxiv.org/abs/2106.02585
  - CLEAR Benchmark: https://clear-benchmark.github.io/

CL - Strategies

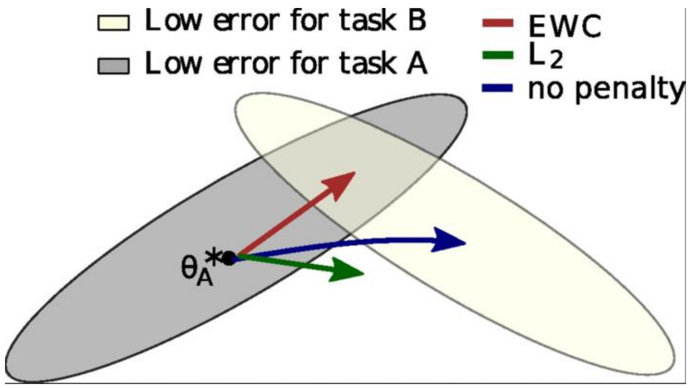
Replay

- Keep a buffer of old samples
- Rehearse old samples



Regularization

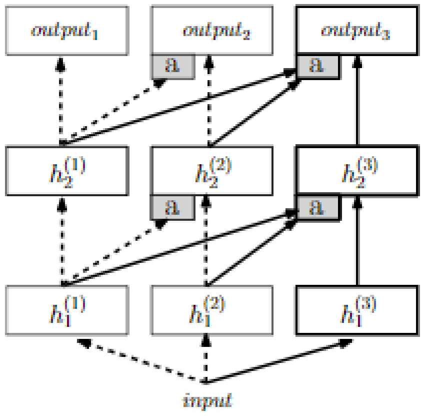
- Regularize the model to balance learning and forgetting



Elastic Weight Consolidation

Architectural

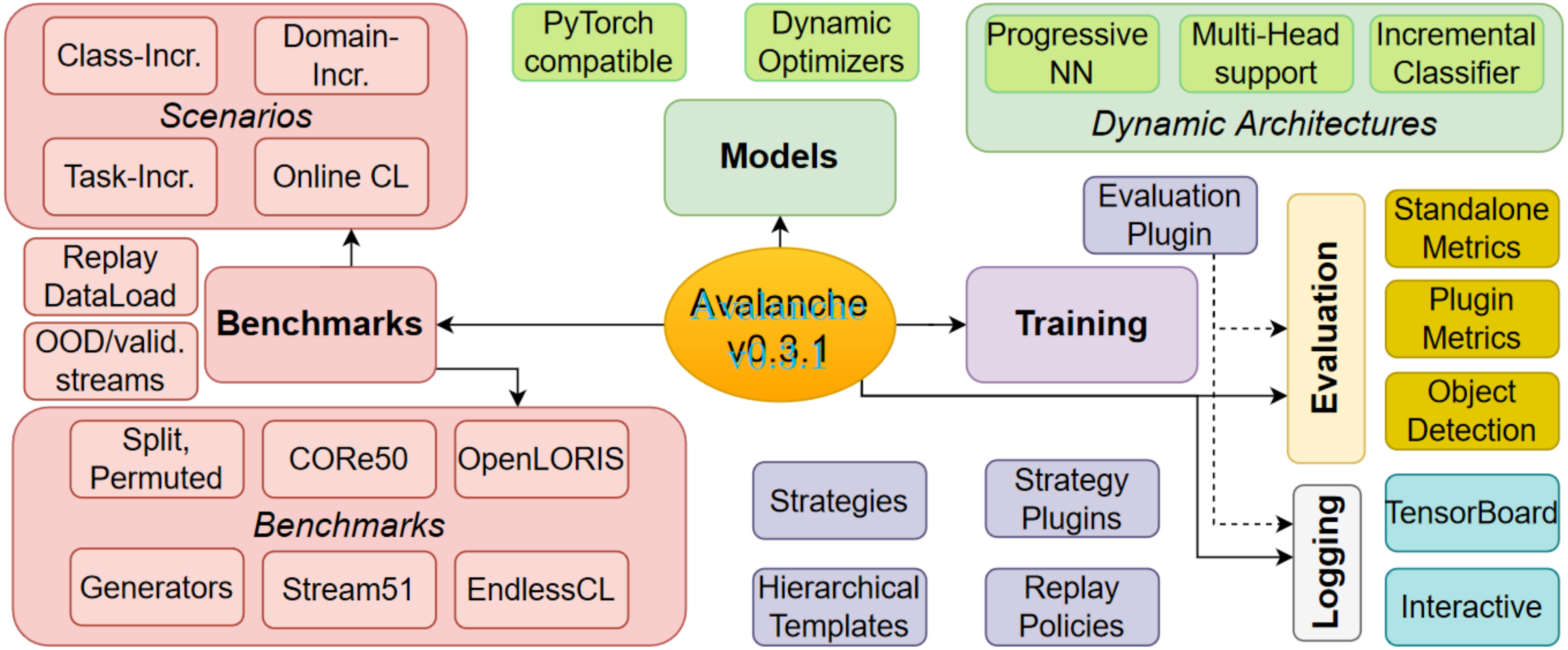
- Expand the model over time with new units/layers



Progressive Neural Networks

Image from https://towardsdatascience.com/reservoir-sampling-for-efficient-stream-processing-97f47f85c11b

What you can do with Avalanche



installing avalanche

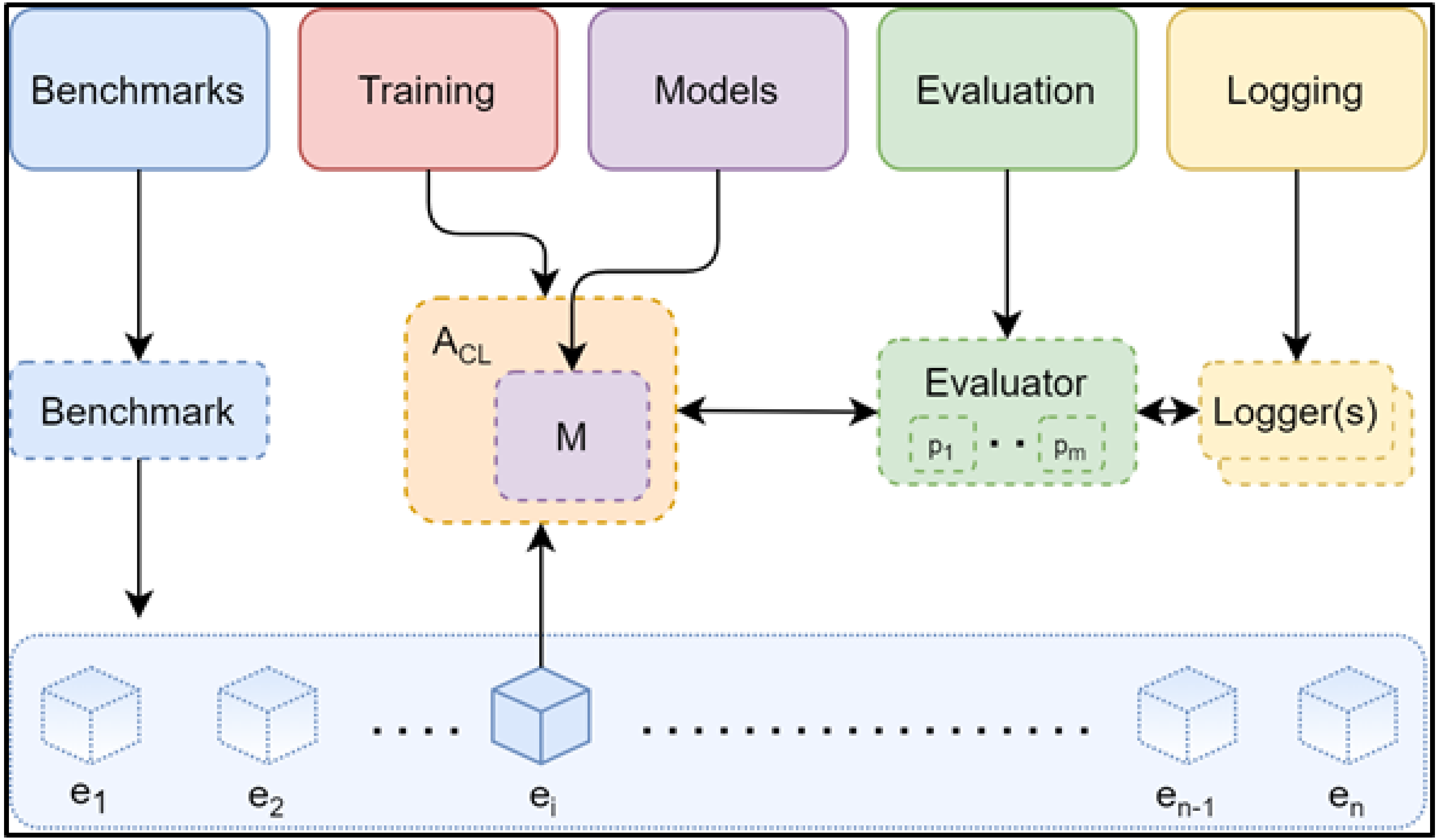
- latest version: 0.3.1, released in Dec 2022
- documentation and tutorials: https://avalanche.continualai.org/
- apidoc: https://avalanche-api.continualai.org/en/v0.3.1/

```
pip install avalanche-lib
```

A Minimal Example

```python
# CL Benchmark Creation
benchmark = PermutedMNIST(n_experiences=3)
train_stream = benchmark.train_stream
test_stream = benchmark.test_stream

# Prepare model, optimizer, criterion (standard pytorch)
model = SimpleMLP(num_classes=10)
optimizer = SGD(model.parameters(), lr=0.001, momentum=0.9)
criterion = CrossEntropyLoss()

# Continual learning strategy
cl_strategy = Naive(
    model, optimizer, criterion,
    train_mb_size=32, train_epochs=2,
    eval_mb_size=32, device=device)

# train and test loop over the stream of experiences
results = []
for train_exp in train_stream:
    cl_strategy.train(train_exp)
    results.append(cl_strategy.eval(test_stream))
```

Avalanche Modules



Continual Learning Streams in Avalanche

In Avalanche, a model learns from a stream of **experiences**:

- **streams** are named sequences (for logging purposes)
- an **experience** contains all the information that is needed for training, evaluation, and logging
  - they have an ID, private and used for logging. Don’t use it during training/evaluation ;)
- additional attributes depending on the **problem type**: a dataset, a list of classes and task labels contained in the experiences…

Supervised Continual Learning Avalanche

In supervised CL:

- Each **experience** provides a dataset `experience.dataset`
- **Datasets** return triplets $< \mathbf{x}, y, t >$
    - $\mathbf{x}$ is the input
    - y is the target class
    - t the task labels, fixed to 0 in task-agnostic scenarios

We give a lot of **freedom** compared to most CL codebases

- classes are not necessarily ordered by experience
- you can have repetitions of classes
- you can have different task labels for samples in the same dataset

Benchmarks

Tutorial: https://avalanche.continualai.org/from-zero-to-hero-tutorial/03_benchmarks

Apidoc: https://avalanche-api.continualai.org/en/v0.3.1/benchmarks.html#

What you need

- Data manipulation: AvalancheDataset
- Definitions of scenarios: benchmark generators
- Benchmarks from the literature

Datasets

Avalanche datasets extend PyTorch datasets:

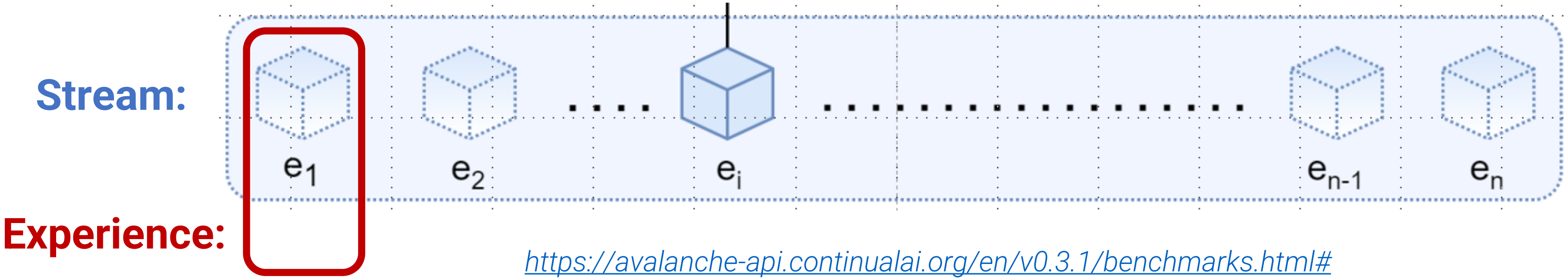
- train/eval **transformations**
- **concatenatenation and subsampling** operations
- `DataAttribute`s keep track of class and task labels
  - they can be used to split datasets by class/task
  - cat/subset operations propagate `DataAttribute`s

You can create benchmarks and implement many replay methods by manipulating Avalanche datasets.

How-to: https://avalanche.continualai.org/how-tos/avalanchedataset
https://avalanche-api.continualai.org/en/v0.3.1/generated/avalanche.benchmarks.utils.AvalancheDataset.html#avalanche.benchmarks.utils.AvalancheDataset

Benchmark, Stream, Experience

- **Benchmark**: a specific instance of a popular setting. It's a collection of streams.
  - Example: SplitMNIST
- **Stream**: a named list of experiences.
  - Example: train/valid/test/ood streams
- **Experience**: the information available at a certain point in time.
  - Example: a dataset, a list of current task/classes, …



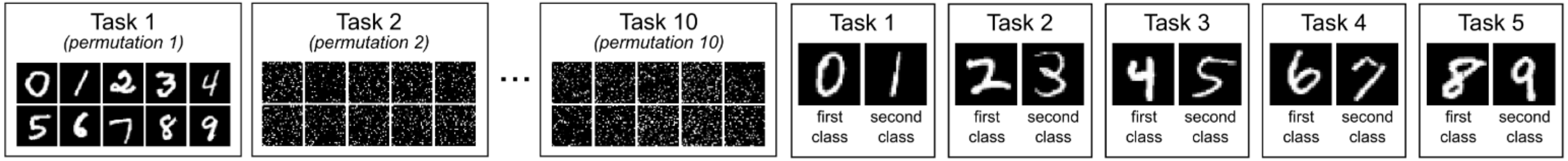
https://avalanche-api.continualai.org/en/v0.3.1/benchmarks.html#

Benchmark – Data Iteration

```
train_stream = benchmark_instance.train_stream
test_stream = benchmark_instance.test_stream

for idx, experience in enumerate(train_stream):
    dataset = experience.dataset

    print('Train dataset contains',
        len(dataset), 'patterns')

    for x, y, t in dataset:
        ...

    test_experience = test_stream[idx]
    cumulative_test = test_stream[:idx+1]
```

Benchmark Generators and Scenarios

- **Scenarios** are abstract problem settings, such as class-incremental, domain-incremental and task-incremental.
- **Benchmark Generators** create a benchmark with specific parameters. Examples:
  - `nc_benchmark`: create a class-incremental benchmark
  - `ni_benchmark`: create a domain-incremental benchmark
  - `dataset_benchmark`: create a supervised CL benchmark from a list of datasets



https://avalanche-api.continualai.org/en/v0.3.1/benchmarks.html#benchmark-generators

Classic Benchmarks

Most common benchmarks from the literature are available and easy to use.

- Reasonable defaults. Usually the most popular configuration in the literature.
- Control over the splits.
- Reproducibility by setting the random seed.

```python
benchmark = SplitMNIST(
    n_experiences=5,
    seed=1,
    return_task_id=False,
    fixed_class_order=[5,0,9, ...],
    train_transform=ToTensor(),
    eval_transform=ToTensor()
)
```

https://avalanche-api.continualai.org/en/v0.3.1/generated/avalanche.benchmarks.classic.SplitMNIST.html#avalanche.benchmarks.classic.SplitMNIST

Models

PyTorch support, architectural, and multitask models.

https://avalanche-api.continualai.org/en/v0.3.1/models.html

Support for pytorch nn.Module

- Avalanche uses pytorch's `nn.Module`
- you can use any model from popular libraries like `torchvision`
- we have additional support for:
  - dynamic modules that change over time
  - multi-task modules where the output depends on task labels
  - update of the optimizer's state (needed for dynamic modules)

```python
# Prepare model, optimizer, criterion (standard pytorch)
model = SimpleMLP(num_classes=10)
optimizer = SGD(model.parameters(), lr=0.001, momentum=0.9)
criterion = CrossEntropyLoss()
```

https://avalanche-api.continualai.org/en/v0.3.1/models.html#

Dynamic Modules

- Dynamic modules grow over time by adding units/layers
  - Incremental classifier
  - Progressive neural network
  - Multi-Task modules
- Only one additional method:
  - `adaptation` takes the new experience and updates the module
  - must be idempontent
  - Don't forget to update the optimizer!

forward, adaptation and optimizer update are called automatically if you use Avalanche training modules

```python
class IncrementalClassifier(DynamicModule):
    """Classifier that adds units whenever new classes are
    encountered."""

    def __init__(
        self,
        in_features,
        initial_out_features=2,
    ):
        super().__init__()
        self.classifier = torch.nn.Linear(in_features, initial_out_features)

    @torch.no_grad()
    def adaptation(self, experience: CLExperience):
        """expand if experience contains unseen classes."""
        in_features = self.classifier.in_features
        old_nclasses = self.classifier.out_features
        new_nclasses = max(self.classifier.out_features, max(curr_classes) + 1)

        # update classifier weights
        if old_nclasses == new_nclasses:
            return
        old_w, old_b = self.classifier.weight, self.classifier.bias
        self.classifier = torch.nn.Linear(in_features, new_nclasses)
        self.classifier.weight[:old_nclasses] = old_w
        self.classifier.bias[:old_nclasses] = old_b

    def forward(self, x, **kwargs):
        return self.classifier(x)
```

Tutorial: https://avalanche.continualai.org/from-zero-to-hero-tutorial/02_models
apidoc: https://avalanche-api.continualai.org/en/v0.3.1/models.html#dynamic-modules

Multi-Task Modules

- Avalanche supports multitask models
- One task labels for each sample
- Standard models, like Multi-head classifiers and PNN are already implemented

- You can use multi-task modules in your models (figure)
- You can also implement your own:
  - Inherit from `MultiTaskModule`
  - `forward_single_task` for examples that have the same task label
  - `MultiTaskModule` implements the `forward` which splits by task the examples.
  - Many multi-task modules also need an incremental `adaptation` step

```python
class MTSimpleMLP(MultiTaskModule):
    """Multi-layer perceptron with multi-head classifier"""

    def __init__(self, input_size=28 * 28, hidden_size=512):
        super().__init__()

        self.features = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(inplace=True),
            nn.Dropout(),
        )
        self.classifier = MultiHeadClassifier(hidden_size)
        self._input_size = input_size

    def forward(self, x, task_labels):
        x = x.contiguous()
        x = x.view(x.size(0), self._input_size)
        x = self.features(x)
        x = self.classifier(x, task_labels)
        return x
```

Training

CL strategies and Avalanche plugins

Tutorial: https://avalanche.continualai.org/from-zero-to-hero-tutorial/04_training
Apidoc: https://avalanche-api.continualai.org/en/v0.3.1/training.html

Contents

- Training methods from the literature
- Definitions of training loops for several CL problems
- A powerful callback systems that links together everything (models, CL methods, evaluation, logging)

High-Level Strategies

- Provides CL methods implementations.
- Different methods can be combined together using plugins.
- You can also implement custom methods.
- train/eval on experiences or streams

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

Replay

- **ReplayPlugin** to use with avalanche strategies
- **Replay buffers** are standalone components
  - You can use them to define a custom replay plugin
  - You can also use them outside Avalanche training loops
- We have also several **dataloaders** to iterate multiple datasets in parallel with or without balancing

```python
from avalanche.training.storage_policy import ReservoirSamplingBuffer
from types import SimpleNamespace

benchmark = SplitMNIST(5, return_task_id=False)
storage_p = ReservoirSamplingBuffer(max_size=30)

print(f"Max buffer size: {storage_p.max_size}," \
      " current size: {len(storage_p.buffer)}")

for i in range(5):
    exp = benchmark.train_stream[i]
    strategy_state = SimpleNamespace(experience=exp)
    storage_p.update(strategy_state)
    print(f"Max buffer size: {storage_p.max_size}," \
          " current size: {len(storage_p.buffer)}")
    print(f"class targets: {storage_p.buffer.targets}\n")
```

Apidoc: https://avalanche-api.continualai.org/en/v0.3.1/training.html#replay-buffers-and-selection-strategies

Plugins

- Avalanche strategies provide a **plugin system**:
  - Methods are called **before/after each event** in the training/evaluation loop
  - Allows to execute code during the loop, **read/write the strategy state**
- **Everything is Avalanche is tied together via the plugin system**
  - CL Training methods are plugins
  - Models forward/adaptation/optimizer update are called inside the training loop and can be overridden by inheritance or adapted with plugins
  - Metric, loggers, and other training utilities are also plugins
- **Advantages**
  - **Compositionality**: You can combine multiple CL training methods together as long as they are compatible (e.g. a regularization method + replay + architectural method)
  - **Reuse**: You can develop a generic plugin and **reuse** it for different domains/scenarios

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

Under the hood: templates

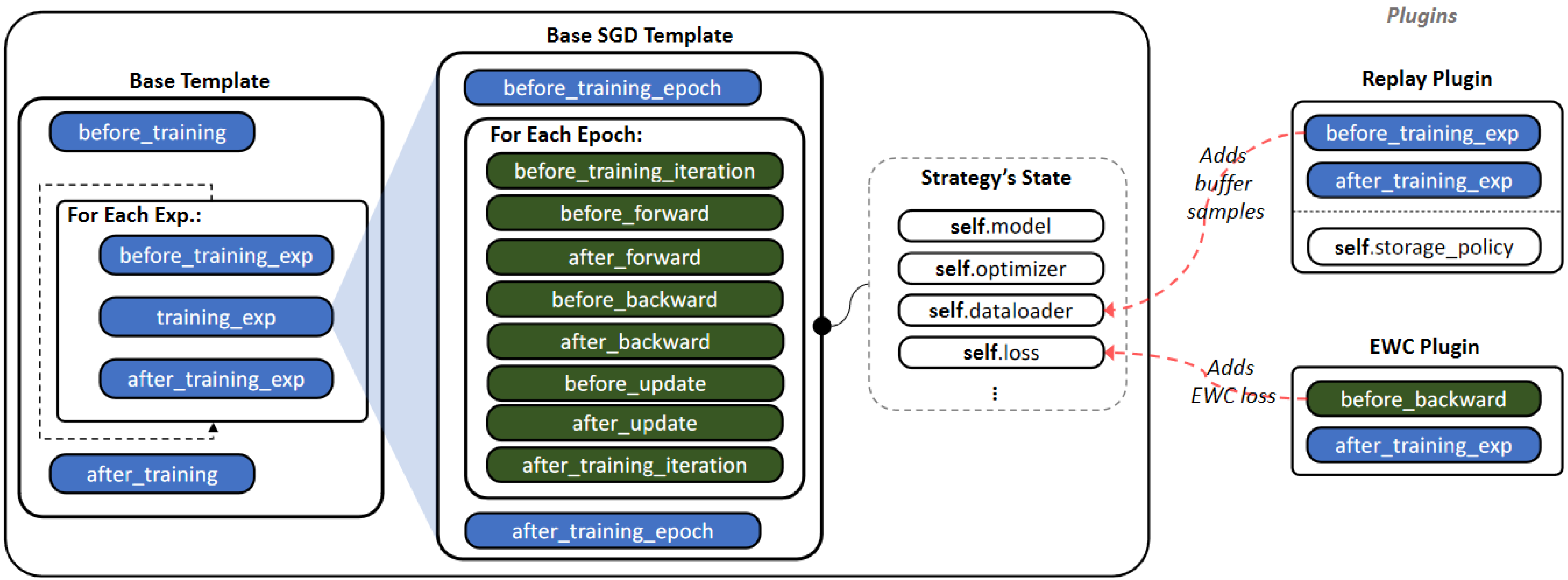


Figure 2: Block diagram of an SGD-based strategy. Replay plugin augments strategy's dataloader while EWC adds a reg. term to the strategy's loss before each update.

Example: Replay

- **Replay methods**:
  - Manage a buffer with old samples, updating it after each experience
  - At each iteration, sample from the new data and buffer jointly
- **In Avalanche**:
  - `before_training_exp` override the default dataloader
    - This works because the dataloader is initialized before this method
  - `after_training_exp` update the replay buffer

```python
class ReplayPlugin(SupervisedPlugin):
    def __init__(self, ...):
        super().__init__()
        ...
        self.storage_policy = ExperienceBalancedBuffer(
            max_size=self.mem_size, adaptive_size=True
        )

    def before_training_exp(self, strategy, **kwargs):
        """Override strategy dataloader"""
        if len(self.storage_policy.buffer) == 0:
            # first experience. We don't use the buffer, no need to change
            # the dataloader.
            return
        batch_size_mem =
        strategy.dataloader = ReplayDataLoader(
            strategy.adapted_dataset,
            self.storage_policy.buffer,
            oversample_small_tasks=True,
            batch_size=strategy.train_mb_size,
            batch_size_mem=self.batch_size_mem,
            task_balanced_dataloader=True,
        )

    def after_training_exp(self, strategy, **kwargs):
        """Update replay buffer."""
        self.storage_policy.update(strategy, **kwargs)
```

Regularization and Architectural Methods

- **Regularization methods**:
  - EWC, LwF, SI, MAS, … (and many hybrid methods)
  - You can implement many regularization methods with just two callbacks:
    - `before_backward` to add your regularization loss
    - `after_training_exp` to update the loss
  - Many of them can be used outside Avalanche by wrapping your training state in a `SimpleNamespace`
- **Architectural methods**:
  - Naive finetuning + a dynamic model (PNN, Multi-head classifier)
  - You don't need a plugin because the adaptation and optimizer's update are already managed by Avalanche loops
  - Easy to use outside of Avalanche loops

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

```
# a multi-head model
model = MTSimpleMLP()
...
# Choose a CL strategy
strategy = Naive(
    model=model,
    ...
)

# train and test loop
for train_task in train_stream:
    strategy.train(train_task)
    strategy.eval(test_stream)
```

Metrics and Evaluation

Tutorial: https://avalanche.continualai.org/from-zero-to-hero-tutorial/05_evaluation
https://avalanche.continualai.org/from-zero-to-hero-tutorial/06_loggers
Apidoc: https://avalanche-api.continualai.org/en/v0.3.1/evaluation.html
https://avalanche-api.continualai.org/en/v0.3.1/logging.html

Contents

- Metrics to evaluate CL methods
- Loggers to store the metrics
- A component to link them with training strategies: EvaluationPlugin

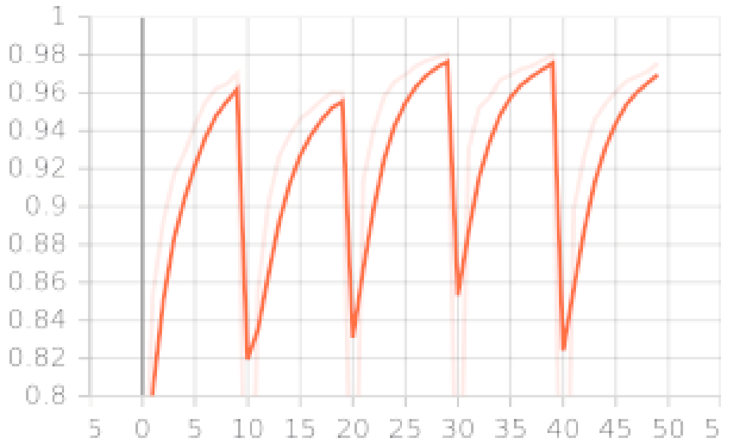
Metrics

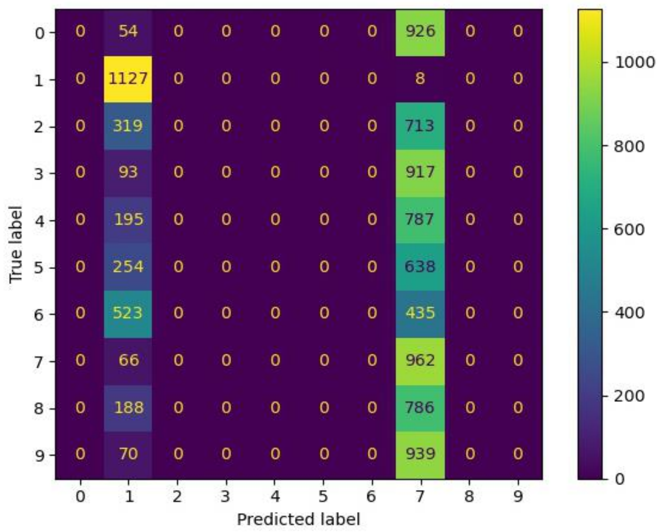
- Available:
  - Accuracy
  - CL-Specific (forgetting, FWT, BWT, …)
  - System usage (memory, CPU, GPU, disk)
- Computed at different granularities (iteration, epoch, experience, stream)

```python
text_logger = TextLogger(open("log.txt", "a"))
interactive_logger = InteractiveLogger()
csv_logger = CSVLogger()
tb_logger = TensorboardLogger()

eval_plugin = EvaluationPlugin(
    accuracy_metrics(
        minibatch=True,
        epoch=True,
        epoch_running=True,
        experience=True,
        stream=True,
    ),
    forgetting_metrics(experience=True, stream=True),
    bwt_metrics(experience=True, stream=True),
    cpu_usage_metrics(epoch=True),
    ram_usage_metrics(every=0.5, experience=True),
    gpu_usage_metrics(args.cuda, every=0.5, minibatch=True),
    loggers=[interactive_logger, text_logger, csv_logger, tb_logger],
    collect_all=True,
)  # collect all metrics (set to True by default)

# CREATE THE STRATEGY INSTANCE (NAIVE)
cl_strategy = Naive(...)

results = []
for i, experience in enumerate(benchmark.train_stream):
    # train returns a dictionary containing last recorded value
    # for each metric.
    res = cl_strategy.train(experience,
                            eval_streams=[benchmark.test_stream])

    # test returns a dictionary with the last metric collected during
    # evaluation on that stream
    results.append(cl_strategy.eval(benchmark.test_stream))

# Dict. Each entry is a (x, metric value) tuple.
all_metrics = cl_strategy.evaluator.get_all_metrics()
print(f"Stored metrics: {list(all_metrics.keys())}")
```

Logging

- Loggers serialize metrics
- Managed by plugin system
- Available:
  - Text logger and terminal
  - Tensorboard
  - CSV
  - Weights and Biases
- You can easily add new loggers





EvaluationPlugin

- Declarative API:
  - Set of metrics to compute
  - Set of loggers for serialization
- Managed by the plugin system
- `train/eval` methods also return a dictionary with all the metrics

```python
text_logger = TextLogger(open("log.txt", "a"))
interactive_logger = InteractiveLogger()
csv_logger = CSVLogger()
tb_logger = TensorboardLogger()

eval_plugin = EvaluationPlugin(
    accuracy_metrics(
        minibatch=True,
        epoch=True,
        epoch_running=True,
        experience=True,
        stream=True,
    ),
    forgetting_metrics(experience=True, stream=True),
    bwt_metrics(experience=True, stream=True),
    cpu_usage_metrics(epoch=True),
    ram_usage_metrics(every=0.5, experience=True),
    gpu_usage_metrics(args.cuda, every=0.5, minibatch=True),
    loggers=[interactive_logger, text_logger, csv_logger, tb_logger],
    collect_all=True,
)  # collect all metrics (set to True by default)

# CREATE THE STRATEGY INSTANCE (NAIVE)
cl_strategy = Naive(...)

results = []
for i, experience in enumerate(benchmark.train_stream):
    # train returns a dictionary containing last recorded value
    # for each metric.
    res = cl_strategy.train(experience,
                            eval_streams=[benchmark.test_stream])

    # test returns a dictionary with the last metric collected during
    # evaluation on that stream
    results.append(cl_strategy.eval(benchmark.test_stream))

# Dict. Each entry is a (x, metric value) tuple.
all_metrics = cl_strategy.evaluator.get_all_metrics()
print(f"Stored metrics: {list(all_metrics.keys())}")
```

Conclusion

where to go for help

- main website: https://avalanche.continualai.org/
- apidoc: https://avalanche-api.continualai.org/en/v0.3.1/
- from zero to hero tutorial: https://avalanche.continualai.org/from-zero-to-hero-tutorial/01_introduction
- have a question or feature requests? https://github.com/ContinualAI/avalanche/discussions
- Found a bug? https://github.com/ContinualAI/avalanche/issues

next: notebooks

https://github.com/AntonioCarta/avalanche-demo

- Avalanche standalone components
- Avalanche end-to-end example