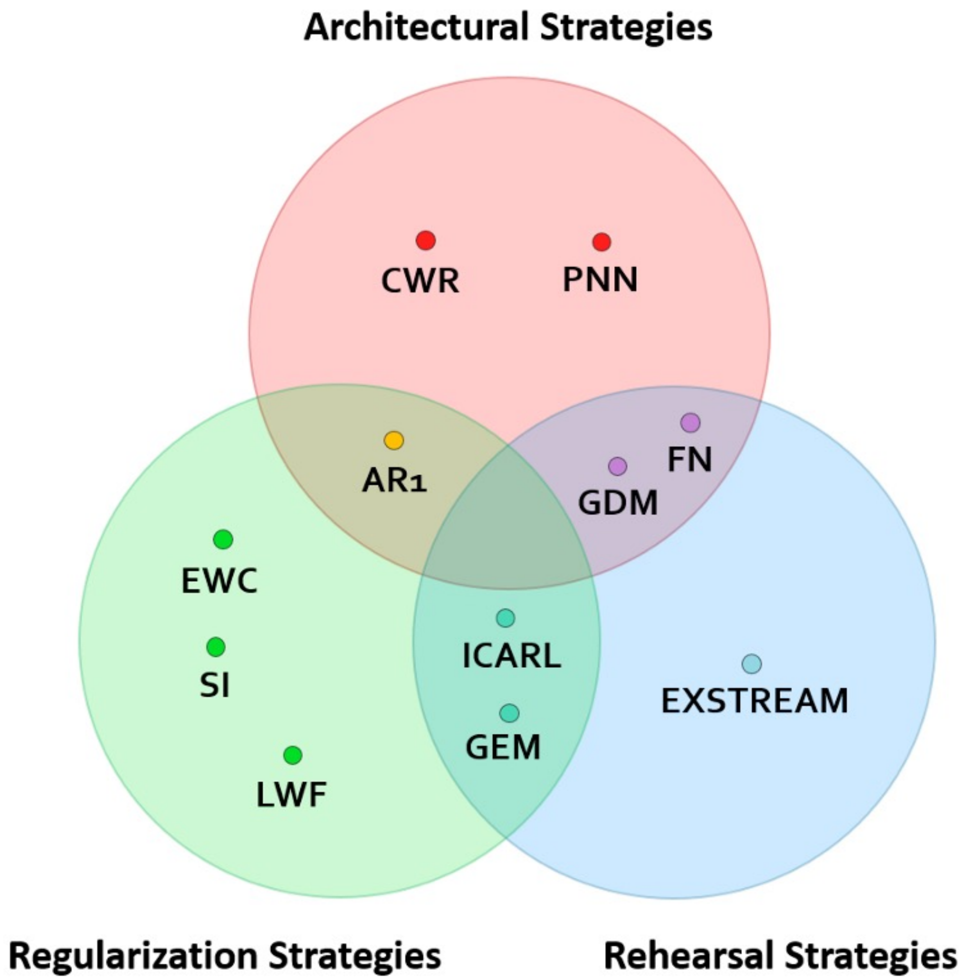
POLITECNICO
MILANO 1863

Continual Learning Strategies and Avalanche

Federico Giannini
Ph.D. Student
DEIB – Politecnico di Milano
federico.giannini@polimi.it

The most popular CL strategies



Baselines:

- **Naïve strategy**

It simply continues the training on the new experience.

- **Cumulative**

During each experience's training concatenate the current experience's training set with all the previous ones.

- **Joint strategy (Offline)**

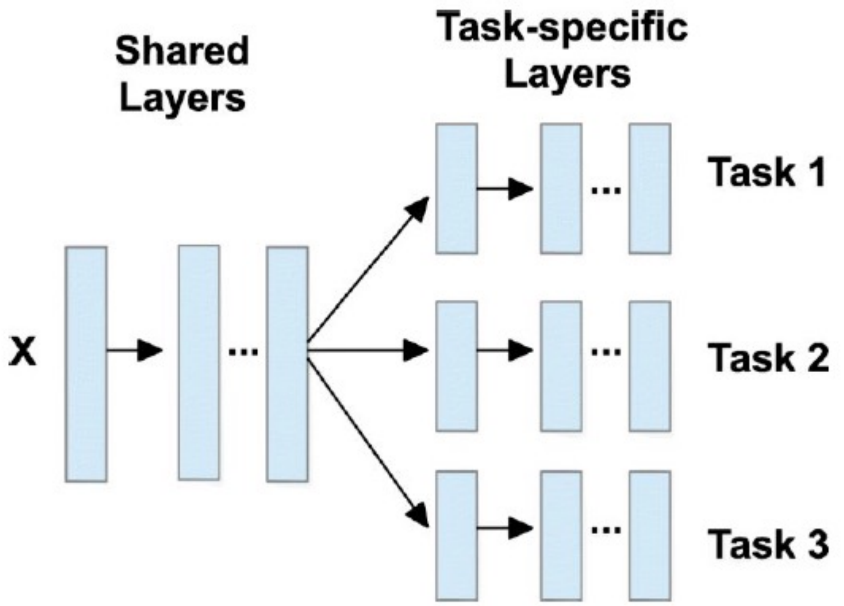
Train the model on all the experiences' training sets together.

Maltoni, Davide, et al. "Continuous learning in single-incremental-task scenarios." Neural Networks 116 (2019): 56-73.

Task Incremental Learning: Multi-Head models

In Task Incremental Learning, a possible design choice is to use a Multi-Head model.

- It adds a new head for each new task (task-specific layers with specific weights).
- θ_o are the parameters associated with old tasks.
- θ_n are the parameters associated with the new task.
- The net has also shared parameters θ_s.
- It requires the task label to select the correct head (some approaches tries to recognize the tasks using an Autoencoder).



Thung, K., et al. "A brief review on multi-task learning." *Multimedia Tools and Applications* 77 (2018): 29705-29725.

Replay: Random Replay

- It uses a fixed-size **Random Memory** (RM) to store a subset of random previous experiences' data points.
- During the training on the i-th experience, it trains the model on the i-th training set shuffled with RM.
- RM contains a random subset of the data points of the previous experiences' training sets.
- After the training, it randomly substitutes some data points with a random subset of the current experience's training set in RM.

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

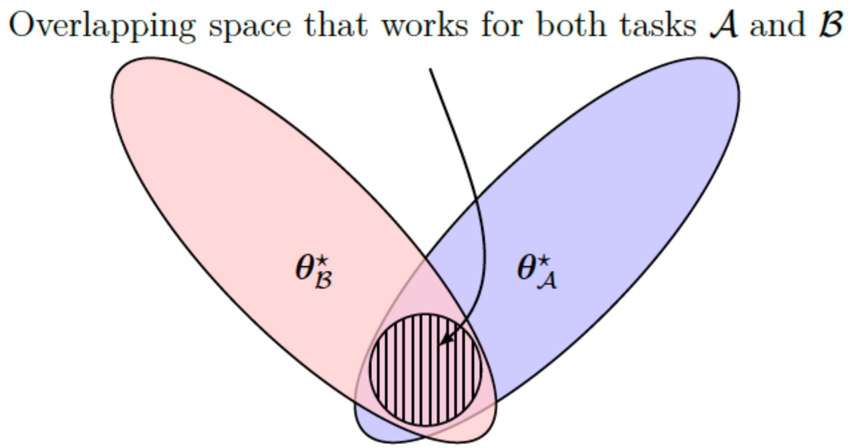
  $$R_{replace} = \begin{cases} \varnothing & \text{if } i == 1 \\ \text{random sample } h \text{ patterns from } RM & \text{otherwise} \end{cases}$$

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

Pellegrini, L., et al. "Latent replay for real-time continual learning." *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020.

Regularization: Elastic Weight Consolidation (EWC) – 1/2

- It adds a regularization term to the loss to penalize changes on important parameters for previous tasks.
- The idea is to search the optimal configuration on the overlapping of different tasks' solution spaces.



Importance index of parameter i for task A

Loss on task B

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \sum_i \frac{\lambda}{2} F_i (\theta_i - \theta^*_{A,i})^2$$

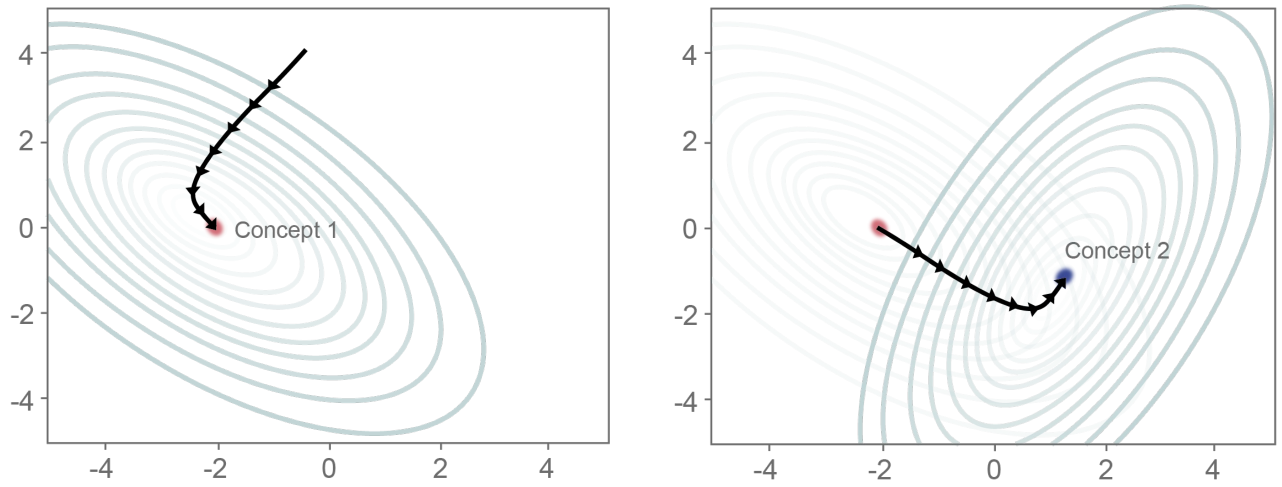
Distance between the current value of parameter i and the optimal one for task A.

Adding the weighted distance means that we want to minimize it for important parameters.

Kirkpatrick, J., et al. Overcoming catastrophic forgetting in neural networks. CoRR abs/1612.00796 (2016)

Regularization: Elastic Weight Consolidation (EWC) – 2/2

- Optimizing very different tasks together could result in not finding a good trade-off:
  - Some tasks may perform well, while others badly.
  - In the worst scenario, we could find a solution that does not fit all the tasks.
- EWC results, in fact, in a tug-of-war over the direction of change of each task.



Kirkpatrick, J., et al. Overcoming catastrophic forgetting in neural networks. *PNAS*, 2017
AICH, Abhishek. Elastic weight consolidation (EWC): Nuts and bolts. *arXiv*, 2021.

Regularization: Learning without Forgetting (LwF) – 1/2

- It is usually applied to Multi-Head models.
- When learning a new task, it uses Knowledge Distillation (KD) to reproduce the outputs of the previous tasks on the new one's data.
- The new training objective is:

$$\min_{\theta_s,\theta_o,\theta_n} \left(\lambda \mathcal{L}_{old}\left(Y_o, \hat{Y}_o\right) + (1-\lambda)\, \mathcal{L}_{new}\left(Y_n, \hat{Y}_n\right) + \mathcal{R}(\theta_s, \theta_o, \theta_n)\right)$$

KD Loss for all the old tasks.
Given an old task o and the current task's data point: Y_o is the ouput returned by the head associated with o of the model at the end of the training on o. $\hat{Y}_o$ is the output returned by the head associated with o of the current model.

Weighted sum's coefficient that indicates the relevance of the KD loss.

Current task's Loss.
Y_n is the real target value of the current task's data point, $\hat{Y}_n$ is the predicted value by the last head.

Regularization term

Li, Z, et al. "Learning without forgetting." *IEEE transactions on pattern analysis and machine intelligence* 40.12 (2017): 2935-2947.

Regularization: Learning without Forgetting (LwF) – 2/2

- The loss function combines two objectives:
  - Reproducing the old tasks' outputs on the previous heads.
  - Learning the new task's classifications.
- It can suffer from the EWC problems if the tasks classification problems are different.
- It does not require storing the previous data (it only uses the current task's dataset).
- It must store the old tasks' models to reproduce their outputs.
- There is also a version that deals with a single head and no task labels.

Li, Z, et al. "Learning without forgetting." *IEEE transactions on pattern analysis and machine intelligence* 40.12 (2017): 2935-2947.

Replay+Regularization: Average Gradient Episodic Memory (AGEM)

- At every training step (mini-batch) it ensures that the average loss on the old tasks does not decrease when learning the current one.
- It randomly chooses a mini-batch from the buffer and computes the gradient g_{ref} on it.
- If computes the gradient g on the current mini-batch.
- If $g^T g_{ref} \geq 0$, it uses g for the gradient update.

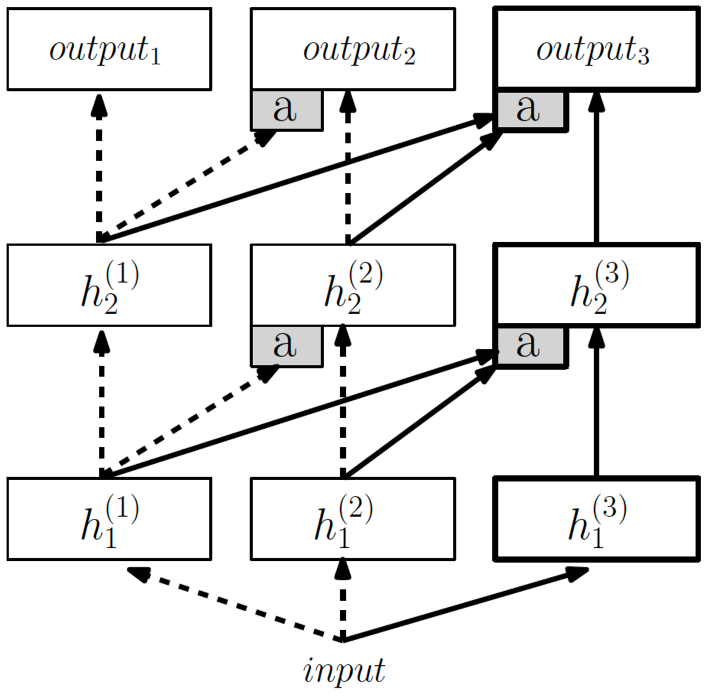
In this ca

- Otherwise, it projects g to have: $g^T g_{ref} = 0$.

A. Chaudhry, et al. Efficient lifelong learning with a-GEM, in: International Conference on Learning Representations, 2019.

Architectural: Progressive Neural Networks (PNN)

- The architecture starts with a single Neural Network (column).
- Whenever a new task appears:
  - It freezes the parameters of the old columns.
  - It adds a new column to the architecture.
- The hidden layer i (i>1) of the column k receives:
  - The output of the hidden layer i-1 of the column k
  - The outputs of the hidden layers i-1 of all the previous column j (j<k).
- It uses transfer learning to combine the knowledge of the previous tasks with that acquired directly from the current task.
- It reuses useful old knowledge and old columns are frozen to avoid CF.
- As the Multi-Head models, it needs to know the current task's label.



Architectural: Copy Weights with Re-init+ (CWR+)

- It's explicitly meant to deal with Class Incremental Scenario.
- The network has **shared weights $\boldsymbol{\Theta}$** that are trained only on the first experience and then frozen.
- During training, it uses the output layer's **temporary weights** tw.
- During inference, it uses the output layer's **consolidated weights** cw.

Algorithm 2 CWR+

```
cw = 0
init Θ̄ random or from a pre-trained model (e.g. ImageNet)
for each training batch B_i:
    expand output layer with s_i neurons for the new classes in B_i
    tw = 0 (for all neurons in the output layer)
    Train the model with SGD on the s_i classes of B_i:
        if B_i = B_1 learn both Θ̄ and tw
        else learn tw while keeping Θ̄ fixed
    for each class j among the s_i classes in B_i:
        cw[j] = tw[j] − avg(tw)
    Test the model by using Θ̄ and cw
```

- On each new experience's training:
  - For each new class, it adds a new neuron to the output layer. It re-init to 0 all the tw.
  - It trains tw on the new experience (while keeping Θ frozen).
  - At the end of the training, for each class j in the experience, it saves the consolidated weights cw (the tw version normalized by subtracting the mean)

Maltoni, Davide, et al. "Continuous learning in single-incremental-task scenarios." Neural Networks 116 (2019): 56-73.

Avalanche

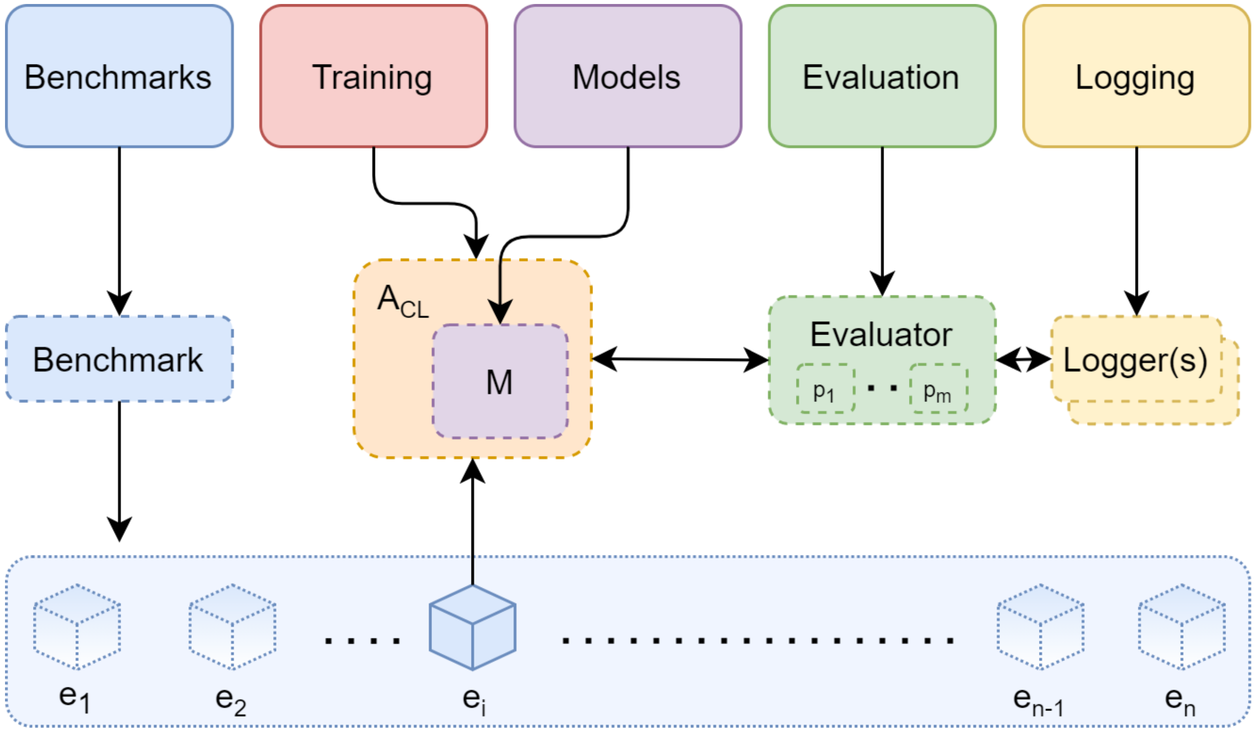
It is an end-to-end library based on Pytorch with the goal of providing a codebase for CL:

- fast prototyping
- training
- reproducible evaluation of algorithms.

Born within ContinualAI with the goal of providing a shared and collaborative open-source codebase.

https://avalanche.continualai.org/

Avalanche's Modules



- **Benchmarks**: It provides data handling. You can generate a data stream from one or more datasets. It contains all the standard benchmarks.
- **Training**: It provides model training. You can implement new CL strategies as well as use a set pre-implemented CL baselines and state-of-the-art algorithms.
- **Evaluation**: It provides all the utilities and metrics that can help in evaluating.
- **Models**: It contains several model architectures and pre-trained models.
- **Logging**: It includes advanced logging and plotting features, including native stdout, file and Tensorboard support.

And now… Let's have some fun!