# Projet

Mohammed ABO ORAIG

February 2026

## 1 Problem Setting : Class-Incremental Learning

### 1.1 Continual Learning Scenario

In this work, we consider the *Class-Incremental Learning* (CIL) setting. In CIL, the training data does not arrive all at once but is instead presented to the model sequentially in a series of learning experiences. Each experience introduces a set of new classes that were not previously observed.

Formally, let the full dataset be split into a sequence of experiences :

$$\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_T\},$$

where each experience $\mathcal{D}_t$ contains samples from a subset of classes $\mathcal{C}_t$, and

$$\mathcal{C}_i \cap \mathcal{C}_j = \emptyset \quad \text{for } i \neq j.$$

At experience $t$, the model has access only to $\mathcal{D}_t$ and possibly a small memory of previous samples. At test time, the model must classify samples among all classes seen so far :

$$\bigcup_{i=1}^{t} \mathcal{C}_i,$$

without being informed about which experience the test sample comes from.

### 1.2 Example : Split CIFAR-10

In our experiments, we use the Split CIFAR-10 benchmark, where the original CIFAR-10 dataset is divided into five experiences, each containing two classes. An example class partition is shown in Table 1.

| Experience | Classes |
|---|---|
| 1 | airplane, automobile |
| 2 | bird, cat |
| 3 | deer, dog |
| 4 | frog, horse |
| 5 | ship, truck |

Table 1 – Example class split for the Split CIFAR-10 benchmark

## 1.3 The Catastrophic Forgetting Problem

A naive approach to class-incremental learning is to train a neural network on each experience independently, using standard supervised learning. However, this approach leads to a severe performance degradation on previously learned classes, a phenomenon known as *catastrophic forgetting*.

The reason for catastrophic forgetting lies in the shared nature of neural network parameters. When the model is trained on a new experience, gradient updates modify the same parameters that were previously responsible for recognizing old classes. As a result, the decision boundaries and internal feature representations learned for old classes are overwritten.

## 1.4 Limitations of Naive Training

More concretely, after learning the first experience, the network develops internal representations that separate the initial classes. When training proceeds on subsequent experiences, these representations shift to accommodate new classes, often at the expense of older ones. Since the original data is no longer available, the model has no mechanism to preserve previously acquired knowledge.

This problem becomes especially challenging in the class-incremental setting, where :

— Task identities are unknown at test time,

— A single classifier must handle all classes,

— Storing the entire past dataset is infeasible.

# 2 Why Naive Classifiers Fail in CIL

## 2.1 Standard Classification with Neural Networks

In standard supervised learning, a neural network classifier is trained to map an input image $x$ to a class label $y \in \{1, \ldots, C\}$. The network consists of :

— A feature extractor $f_\theta(\cdot)$, parameterized by $\theta$,

— A linear classifier $W \in R^{C \times d}$.

The predicted class probabilities are obtained using a softmax function :

$$p(y = c \mid x) = \frac{\exp(W_c^\top f_\theta(x))}{\sum_{k=1}^{C} \exp(W_k^\top f_\theta(x))}.$$

Training is performed by minimizing the cross-entropy loss over a dataset containing samples from all classes.

## 2.2 What Changes in Class-Incremental Learning

In class-incremental learning, the training dataset at experience $t$, denoted $\mathcal{D}_t$, contains samples only from the newly introduced classes $\mathcal{C}_t$. Crucially, samples from previous classes $\bigcup_{i=1}^{t-1} \mathcal{C}_i$ are no longer fully available.

Despite this, the classifier must still produce predictions over the union of all classes seen so far :

$$\bigcup_{i=1}^{t} \mathcal{C}_i.$$

This creates a mismatch between the training objective and the evaluation objective.

## 2.3 Classifier Bias Toward New Classes

When training with cross-entropy loss on only the current experience $\mathcal{D}_t$, the loss function encourages the model to increase the logits corresponding to the new classes, while implicitly suppressing the logits of old classes.

Since no samples from old classes are present, there is no gradient signal enforcing their preservation. As a result :

— The weights associated with old classes become poorly calibrated,

— The classifier becomes biased toward recently learned classes,

— Predictions increasingly favor new classes, even for old-class inputs.

This phenomenon is known as *classifier bias.*

## 2.4 Representation Drift

In addition to classifier bias, the feature extractor itself suffers from *representation drift.* As the model is trained on new classes, the feature space changes to better separate the new classes. Consequently, feature representations of old-class samples (if they were to be recomputed) shift over time.

Let $f_{\theta_{t-1}}(x)$ denote the feature representation of a sample $x$ after experience $t-1$, and $f_{\theta_t}(x)$ the representation after experience $t$. In general :

$$f_{\theta_{t-1}}(x) \neq f_{\theta_t}(x),$$

even if $x$ belongs to an old class.

This drift causes old decision boundaries to become invalid, further degrading performance.

## 2.5 Illustrative Example

Consider a simple example in which a model first learns to classify images of airplanes and automobiles. The learned feature space separates these two classes effectively.

When new classes such as birds and cats are introduced, the feature extractor adapts to separate these new categories. However, since airplane and automobile images are no longer available, their feature representations are not constrained and may collapse or overlap with new classes.

As a result, the model may incorrectly classify airplane images as birds or automobiles as cats.

## 2.6 Limitations of Softmax-Based Classification

Softmax-based classifiers implicitly assume that all classes are observed simultaneously during training. In the class-incremental setting, this assumption is violated.

Specifically :

— The normalization term in the softmax includes classes that are not present in the current training data,

— The classifier weights for old classes are not updated using representative data,

— The learned decision boundaries are unstable across experiences.

These limitations make standard softmax classifiers unsuitable for class-incremental learning without additional mechanisms.

# 3 Elastic Weight Consolidation (EWC)

## 3.1 Motivation

In continual learning, a model is trained sequentially on a series of tasks or experiences. At time step $t$, the model has access only to the data of the current task $\mathcal{D}_t$, while data from previous tasks $\mathcal{D}_1, \ldots, \mathcal{D}_{t-1}$ are no longer available.

When a standard neural network is trained on a new task using gradient descent, its parameters are updated to minimize the loss on the current data :

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_t(\theta).$$

This update does not take into account whether certain parameters were crucial for solving previous tasks. As a result, parameters that were important for old tasks may change significantly, leading to a sudden drop in performance on those tasks.

## 3.2 Core Idea of EWC

Elastic Weight Consolidation addresses catastrophic forgetting by introducing the following key idea :

*Not all parameters are equally important for previously learned tasks.*

EWC identifies which parameters are crucial for old tasks and restricts their movement when learning new tasks. Instead of storing old data (as in replay-based methods), EWC stores *importance information* about the parameters.

This makes EWC :

— Memory-efficient,

— Data-free (no exemplar storage),

— Suitable for privacy-sensitive scenarios.

EWC makes important parameters *stiff* (hard to change), while allowing unimportant parameters to remain *plastic*. This stability–plasticity balance is the central principle behind Elastic Weight Consolidation.

## 3.3 Problem Formulation and Notation

We consider a continual learning setting in which a model is trained sequentially on a series of tasks or experiences :

$$\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_T.$$

Each task $\mathcal{T}_t$ is associated with a dataset :

$$\mathcal{D}_t = \{(x_i^{(t)}, y_i^{(t)})\}_{i=1}^{N_t},$$

where $x_i^{(t)} \in \mathcal{X}$ denotes the input (e.g., an image) and $y_i^{(t)} \in \mathcal{Y}_t$ denotes the corresponding label.

The model is parameterized by a vector of parameters :

$$\theta = (\theta_1, \theta_2, \ldots, \theta_P),$$

where $P$ is the total number of trainable parameters.

### 3.3.1 Sequential Training Constraint

At training time $t$, the learner :

— Has access only to the current dataset $\mathcal{D}_t$,

— Cannot revisit datasets $\mathcal{D}_1, \ldots, \mathcal{D}_{t-1}$,

— Must preserve performance on all previously learned tasks.

At test time, the model is evaluated on all tasks seen so far :

$$\mathcal{T}_1 \cup \mathcal{T}_2 \cup \cdots \cup \mathcal{T}_t,$$

without being told which task an input belongs to.

This setting corresponds to the class-incremental learning scenario when task identities are not available at inference.

### 3.3.2 Objective Without Continual Learning Constraints

If tasks were independent and forgetting was not a concern, training on task $t$ would consist of minimizing the empirical risk :

$$\mathcal{L}_t(\theta) = E_{(x,y)\sim\mathcal{D}_t}\left[\ell(f_\theta(x), y)\right],$$

where :

— $f_\theta$ is the neural network,

— $\ell(\cdot, \cdot)$ is a standard loss function (e.g., cross-entropy).

However, minimizing $\mathcal{L}_t(\theta)$ alone leads to catastrophic forgetting, since it ignores the constraints imposed by previous tasks.

### 3.3.3 Continual Learning Objective

In continual learning, the objective at time $t$ should :

— Minimize the loss on the current task $\mathcal{T}_t$,

— Preserve knowledge acquired from previous tasks $\mathcal{T}_1, \ldots, \mathcal{T}_{t-1}$.

Formally, this corresponds to solving a constrained optimization problem :

$$\min_\theta \; \mathcal{L}_t(\theta) \quad \text{subject to} \quad f_\theta \approx f_{\theta_k^*} \text{ for all } k < t,$$

where $\theta_k^*$ denotes the optimal parameters after training on task $\mathcal{T}_k$.

EWC proposes a principled relaxation of this constraint by introducing a task-dependent regularization term, which will be derived in the next section.

## 3.4 Bayesian Interpretation of Continual Learning

EWC is derived from a Bayesian perspective on learning. In this view, learning a task corresponds to updating a posterior distribution over the model parameters.

### 3.4.1 Bayesian Learning for a Single Task

Given a dataset $\mathcal{D}_t$, Bayes' rule gives the posterior distribution over parameters :

$$p(\theta \mid \mathcal{D}_t) = \frac{p(\mathcal{D}_t \mid \theta)\, p(\theta)}{p(\mathcal{D}_t)}.$$

Here :

— $p(\theta)$ is the prior over parameters,

— $p(\mathcal{D}_t \mid \theta)$ is the likelihood,

— $p(\theta \mid \mathcal{D}_t)$ is the posterior after observing the data.

In practice, neural networks are trained by maximizing the log-posterior, which is equivalent to minimizing the negative log-posterior :

$$-\log p(\theta \mid \mathcal{D}_t) = -\log p(\mathcal{D}_t \mid \theta) - \log p(\theta) + \text{const}.$$

This shows that Bayesian learning naturally leads to a loss function composed of :

— A data-fitting term (likelihood),

— A regularization term (prior).

### 3.4.2 Sequential Bayesian Updating

In a continual learning setting, tasks arrive sequentially. After completing task $\mathcal{T}_{t-1}$, the posterior becomes :

$$p(\theta \mid \mathcal{D}_{1:t-1}).$$

When learning task $\mathcal{T}_t$, this posterior should act as the prior :

$$p(\theta \mid \mathcal{D}_{1:t}) \propto p(\mathcal{D}_t \mid \theta)\, p(\theta \mid \mathcal{D}_{1:t-1}).$$

Thus, learning a new task should not overwrite previous knowledge, but refine it.

### 3.4.3 Intractability in Neural Networks

For modern neural networks, the exact posterior $p(\theta \mid \mathcal{D}_{1:t})$ is intractable :

— The parameter space is extremely high-dimensional,

— The likelihood is highly non-linear,

— Exact Bayesian inference is computationally infeasible.

EWC addresses this by approximating the posterior with a simpler distribution.

### 3.4.4 Laplace Approximation of the Posterior

EWC approximates the posterior after task $\mathcal{T}_{t-1}$ using a Gaussian distribution centered at the maximum a posteriori (MAP) estimate :

$$p(\theta \mid \mathcal{D}_{1:t-1}) \approx \mathcal{N}(\theta^*_{t-1}, \Sigma),$$

where :

— $\theta^*_{t-1}$ is the parameter vector obtained after training on task $t-1$,

— $\Sigma$ is a covariance matrix encoding parameter uncertainty.

For computational efficiency, EWC assumes a diagonal covariance matrix :

$$\Sigma = \mathrm{diag}(\sigma_1^2, \dots, \sigma_P^2).$$

This assumption implies that parameters are treated as independent, greatly simplifying the method.

### 3.4.5 Interpretation

Under this approximation :

— Parameters with low variance (small $\sigma_i^2$) are *important* and should not change much,

— Parameters with high variance are less constrained and can adapt to new tasks.

The challenge now reduces to estimating how important each parameter is for the previous tasks. Elastic Weight Consolidation proposes to estimate this importance using the Fisher Information Matrix, which is introduced in the next section.

## 3.5 Fisher Information Matrix and Parameter Importance

The central question in Elastic Weight Consolidation is :

*Which parameters are important for solving previously learned tasks ?*

EWC answers this question using the **Fisher Information Matrix** (FIM), a classical concept from information theory and statistics.

### 3.5.1 Definition of the Fisher Information Matrix

For a probabilistic model $p(y \mid x, \theta)$, the Fisher Information Matrix is defined as :

$$F(\theta) = E_{(x,y)\sim\mathcal{D}} \left[ \nabla_\theta \log p(y \mid x, \theta) \, \nabla_\theta \log p(y \mid x, \theta)^\top \right].$$

Intuitively, the Fisher Information measures how sensitive the model's predictions are to changes in each parameter.

### 3.5.2 Diagonal Approximation

Because neural networks have millions of parameters, storing the full Fisher matrix is infeasible. EWC therefore uses a diagonal approximation :

$$F \approx \mathrm{diag}(F_1, F_2, \dots, F_P),$$

where each $F_i$ corresponds to the importance of parameter $\theta_i$.

This approximation assumes parameter independence, which is a practical but effective simplification.

### 3.5.3 Interpretation of Parameter Importance

The diagonal Fisher element $F_i$ can be interpreted as :

— Large $F_i$ : small changes in $\theta_i$ strongly affect the output distribution $\Rightarrow$ parameter is important.

— Small $F_i$ : changes in $\theta_i$ have little effect $\Rightarrow$ parameter is less important.

Thus, the Fisher Information provides a principled, data-driven measure of parameter importance.

### 3.5.4 Estimating the Fisher Information in Practice

In practice, the Fisher Information is estimated after completing training on task $\mathcal{T}_{t-1}$. For classification tasks with cross-entropy loss, the empirical Fisher can be computed as :

$$F_i \approx \frac{1}{|\mathcal{D}_{t-1}|} \sum_{(x,y)\in\mathcal{D}_{t-1}} \left( \frac{\partial \log p(y \mid x, \theta^*_{t-1})}{\partial \theta_i} \right)^2 .$$

This computation requires :

- — A forward pass to compute predictions,
- — A backward pass to compute gradients,
- — Squaring and averaging gradients for each parameter.

The resulting Fisher values are stored and reused during future training phases.

### 3.5.5 Why Fisher Information Makes Sense

The Fisher Information arises naturally from the second-order Taylor expansion of the log-likelihood around the optimum. Under mild assumptions, it approximates the curvature of the loss surface :

$$\mathcal{L}(\theta) \approx \mathcal{L}(\theta^*) + \frac{1}{2}(\theta - \theta^*)^\top F(\theta - \theta^*).$$

This quadratic form justifies the use of Fisher Information as a measure of how "stiff" or "elastic" each parameter should be when learning new tasks.

## 3.6 Elastic Weight Consolidation Loss Function

### 3.6.1 Derivation of the EWC Penalty

Recall from Step 3 that, under a Bayesian perspective, the posterior for parameters after task $t-1$ acts as a prior for task $t$ :

$$p(\theta \mid \mathcal{D}_{1:t}) \propto p(\mathcal{D}_t \mid \theta)\, p(\theta \mid \mathcal{D}_{1:t-1}).$$

Using the Laplace approximation with a diagonal Fisher (Step 4), the negative log-prior is :

$$-\log p(\theta \mid \mathcal{D}_{1:t-1}) \approx \sum_{i=1}^{P} \frac{1}{2} F_i (\theta_i - \theta_i^*)^2 + \text{const},$$

where :

- — $F_i$ is the Fisher Information for parameter $\theta_i$,
- — $\theta_i^*$ is the parameter value after the previous task.

### 3.6.2 Total EWC Loss

The total loss when training on task $t$ becomes :

$$\mathcal{L}_{EWC}(\theta) = \mathcal{L}_t(\theta) + \frac{\lambda}{2} \sum_{i=1}^{P} F_i (\theta_i - \theta_i^*)^2,$$

where :

— $\mathcal{L}_t(\theta)$ is the standard task loss (e.g., cross-entropy),
— The second term is the **EWC regularization term**,
— $\lambda$ is a hyperparameter controlling the strength of the regularization.

### 3.6.3 Intuition

The EWC penalty can be interpreted as attaching a spring to each parameter :
— The spring is stiffer for important parameters (large $F_i$),
— The spring is looser for unimportant parameters (small $F_i$),
— Gradient updates are slowed down proportionally to parameter importance.

This mechanism preserves knowledge about previous tasks while allowing flexibility to learn new tasks.

### 3.6.4 Illustrative Example

Suppose a neural network has three parameters after task 1 :

$$\theta^* = [0.5, 1.2, -0.3], \quad F = [10, 1, 0.1].$$

During training on task 2, the EWC penalty for each parameter is :

$$\frac{\lambda}{2} F_i(\theta_i - \theta_i^*)^2 = \frac{\lambda}{2}[10(\theta_1 - 0.5)^2 + 1(\theta_2 - 1.2)^2 + 0.1(\theta_3 + 0.3)^2].$$

Interpretation :
— Parameter $\theta_1$ is highly important and will move very little,
— Parameter $\theta_3$ is almost irrelevant and can freely adapt,
— Parameter $\theta_2$ is moderately constrained.

### 3.6.5 Connection to Implementation

In your Avalanche EWC code :
— `ewc_lambda` corresponds to $\lambda$,
— Fisher Information $F_i$ is computed internally after each task,
— The total loss `criterion + EWC penalty` is minimized using the optimizer (SGD or Adam).

By using this loss, EWC enforces the stability–plasticity tradeoff across tasks, preventing catastrophic forgetting while still learning new information.

## 3.7 Training Loop and Metrics in EWC

### 3.7.1 Incremental Training Procedure

Training with EWC proceeds sequentially, task by task. At experience $t$, the model performs the following steps :

1. **Receive current task data $\mathcal{D}_t$**, containing only new task samples.

2. **Retrieve stored parameters and Fisher information** from previous tasks :
$$\theta^* = \theta^*_{t-1}, \quad F = F_{t-1}.$$
3. **Compute the total loss** for each mini-batch :
$$\mathcal{L}_{EWC}(\theta) = \mathcal{L}_t(\theta) + \frac{\lambda}{2}\sum_i F_i(\theta_i - \theta_i^*)^2.$$
4. **Update parameters** using gradient descent (e.g., SGD or Adam).
5. **After training**, compute the Fisher Information for the current task and update stored values :
$$F \leftarrow F + F_t.$$
6. **Repeat** for all subsequent tasks.

### 3.7.2 Illustrative Example

Suppose a model has three parameters $\theta = [\theta_1, \theta_2, \theta_3]$, and after task 1 we have :
$$\theta^* = [0.5, 1.2, -0.3], \quad F = [10, 1, 0.1].$$
When training on task 2 with cross-entropy loss on new data :
- The optimizer updates $\theta$ to reduce the new task loss,
- The EWC term penalizes large deviations from $\theta^*$, proportionally to $F$,
- Parameter $\theta_1$ barely changes (highly important),
- Parameter $\theta_3$ adapts freely (low importance).

This illustrates how EWC balances **stability** (old tasks) with **plasticity** (new tasks).

### 3.7.3 Evaluation Metrics

As in iCaRL, we can track continual learning metrics :
- **Accuracy (Stream)** : classification accuracy on all tasks seen so far after training on task $t$,
- **Forgetting Measure (FM)** : quantifies how much performance drops on previous tasks,
- **Average Accuracy (AA)** : average performance across all tasks at the end of training,
- **Backward Transfer (BWT)** : captures influence of new learning on old tasks.

### 3.7.4 Example Intuition

Suppose the model sees tasks sequentially and achieves the following stream accuracy after each task :
$$\text{Task 1} : 0.85, \quad \text{Task 2} : 0.80, \quad \text{Task 3} : 0.78.$$
Observations :
- Accuracy decreases slightly but not catastrophically, indicating that EWC successfully preserved old knowledge,
- Forgetting Measure remains low due to the regularization term,
- New tasks are learned without completely overwriting old tasks.

### 3.7.5 Connection to Implementation

In Avalanche :

— `EWC` strategy automates the storage of $\theta^*$ and $F$,

— `ewc_lambda` controls the strength of the regularization term,

— `train_mb_size`, `train_epochs`, and optimizer choice behave as in standard supervised training.

This ensures that, although EWC is conceptually more complex than naive training, its practical usage is straightforward.

## 3.8 Practical Considerations and Hyperparameters in EWC

### 3.8.1 The Regularization Strength $\lambda$

The hyperparameter $\lambda$ controls the trade-off between :

— **Stability** : preserving old knowledge,

— **Plasticity** : learning new tasks effectively.

**Effect of $\lambda$**

— $\lambda$ too small : old knowledge is forgotten (EWC behaves like naive fine-tuning),

— $\lambda$ too large : new task learning is overly constrained, resulting in underfitting.

**Typical values** Empirically, $\lambda$ is often chosen in the range $10^2 - 10^5$ depending on :

— Network size,

— Dataset complexity,

— Number of tasks.

### 3.8.2 Optimizer Choice

EWC can be used with standard optimizers such as :

— **SGD with momentum** : often preferred for stability in continual learning,

— **Adam** : allows faster convergence but may require tuning $\lambda$ carefully.

**Intuition** The optimizer affects how the gradients are applied to parameters with varying importance (Fisher values) :

— SGD : small, controlled updates help respect the EWC penalty,

— Adam : adaptive updates may accidentally violate parameter constraints if $\lambda$ is too small.

### 3.8.3 Batch Size and Epochs

The choice of mini-batch size and number of epochs affects :

— Stability of Fisher estimation,

— Convergence on the new task,

— Overall performance across tasks.

Practical guidelines :

— Use moderate batch sizes (e.g., 32-128) for stable Fisher estimation,

— Train for enough epochs to ensure the new task is learned without violating the old task constraints.

### 3.8.4 Scaling to Multiple Tasks

For $t > 2$ tasks, EWC can accumulate Fisher Information from previous tasks :

$$\mathcal{L}^{(t)}_{EWC}(\theta) = \mathcal{L}_t(\theta) + \frac{\lambda}{2}\sum_i \sum_{k=1}^{t-1} F_i^{(k)}(\theta_i - \theta_i^{*(k)})^2.$$

In practice :

— Storing a separate $\theta^*$ and $F$ for each task is feasible for a small number of tasks,

— For many tasks, a running average of Fisher values can reduce memory usage.

### 3.8.5 Comparison to Replay-Based Methods

**Memory Requirement**

— EWC : stores only parameter values and Fisher diagonals ($O(P)$ memory),

— iCaRL / replay : stores actual exemplars ($O(M)$ memory, often much larger).

**Performance Trend**

— EWC : stability depends heavily on $\lambda$ and Fisher estimation ; forgetting is reduced but not eliminated,

— Replay-based methods : more robust retention of old classes but at the cost of memory usage.

**Flexibility** EWC can be combined with other techniques, e.g., small replay buffers, to improve performance on long sequences of tasks.

### 3.8.6 Illustrative Example

Consider training EWC on CIFAR-10 with 5 incremental experiences :

— $\lambda = 1000$, SGD with momentum 0.9, batch size 32, 50 epochs,

— Fisher computed at the end of each experience,

— Accuracy (Stream) decreases slightly after each experience but remains above 70% for previous tasks.

This demonstrates that, with proper hyperparameter tuning, EWC provides a practical and memory-efficient solution for class-incremental learning.

## 3.9 Full EWC Algorithm and Flowchart

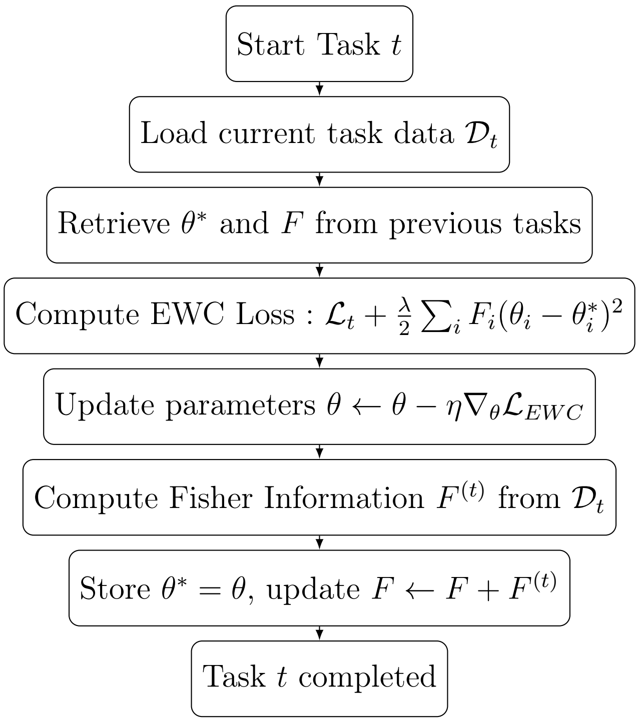
### 3.9.1 Algorithm Description

The following algorithm summarizes the Elastic Weight Consolidation procedure in a clear, step-by-step manner.

**Algorithm 1** Elastic Weight Consolidation (EWC) for Continual Learning

```
Require: Sequential tasks $\mathcal{T}_1, \ldots, \mathcal{T}_T$, learning rate $\eta$, regularization strength $\lambda$
 1: Initialize parameters $\theta$
 2: for $t = 1$ to $T$ do
 3:     Receive dataset $\mathcal{D}_t$ for task $\mathcal{T}_t$
 4:     if $t > 1$ then
 5:         Retrieve stored parameters $\theta^* = \theta^*_{t-1}$ and Fisher $F = F_{t-1}$
 6:     end if
 7:     Train model on $\mathcal{D}_t$ by minimizing :
            $$\mathcal{L}_{EWC}(\theta) = \mathcal{L}_t(\theta) + \frac{\lambda}{2} \sum_i F_i(\theta_i - \theta_i^*)^2$$
 8:     Update $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_{EWC}(\theta)$
 9:     Compute Fisher Information $F^{(t)}$ from current task :
            $$F_i^{(t)} \approx \frac{1}{|\mathcal{D}_t|} \sum_{(x,y) \in \mathcal{D}_t} \left( \frac{\partial \log p(y|x,\theta)}{\partial \theta_i} \right)^2$$
10:     Store $\theta^* = \theta$, $F = F + F^{(t)}$ for future tasks
11: end for
12: Return final parameters $\theta$
```

### 3.9.2 Flowchart of EWC



Start Task $t$
Load current task data $\mathcal{D}_t$
Retrieve $\theta^*$ and $F$ from previous tasks
Compute EWC Loss : $\mathcal{L}_t + \frac{\lambda}{2} \sum_i F_i(\theta_i - \theta_i^*)^2$
Update parameters $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_{EWC}$
Compute Fisher Information $F^{(t)}$ from $\mathcal{D}_t$
Store $\theta^* = \theta$, update $F \leftarrow F + F^{(t)}$
Task $t$ completed

FIGURE 1 – Flowchart of the Elastic Weight Consolidation procedure for a single task.

### 3.9.3 Explanation of Flowchart

— The model receives new task data at each experience.

— The previous parameters and Fisher Information are retrieved to compute the EWC penalty.

— The total loss combines the new task loss with the regularization term.

— After parameter updates, the Fisher Information is recomputed and stored for future tasks.

— This loop continues sequentially for all tasks, ensuring old knowledge is preserved.

## 3.10 Summary, Strengths, Limitations, and Key Takeaways of EWC

### 3.10.1 Summary of EWC

Elastic Weight Consolidation (EWC) is a regularization-based method for continual learning that :

— Treats previously learned tasks as a prior on the parameters (Bayesian perspective),

— Quantifies parameter importance using the Fisher Information Matrix,

— Introduces a quadratic penalty to prevent important parameters from changing during training on new tasks,

— Operates sequentially without storing old data, making it memory-efficient.

### 3.10.2 Strengths

- **Memory Efficiency :** Only stores parameter snapshots and Fisher diagonals, avoiding the need for large replay buffers.
- **Principled Bayesian Foundation :** Provides a theoretically justified way to measure parameter importance.
- **Simplicity and Compatibility :** Can be combined with any optimizer or neural network architecture.
- **Mitigates Catastrophic Forgetting :** Preserves performance on old tasks, especially for moderate sequences of tasks.

### 3.10.3 Limitations

- **Approximation of Fisher :** Diagonal approximation ignores parameter correlations, limiting effectiveness for complex networks.
- **Limited Plasticity :** Strong regularization ($\lambda$ large) may hinder learning new tasks.
- **Scaling to Many Tasks :** Accumulating Fisher Information over many tasks can lead to overly restrictive updates or require memory for multiple snapshots.
- **Less Effective than Replay-Based Methods :** For long sequences of tasks or highly dissimilar tasks, exemplar-based methods like iCaRL may outperform EWC.

### 3.10.4 Practical Recommendations

- Carefully tune $\lambda$ to balance stability and plasticity.
- Use moderate batch sizes and sufficient epochs to ensure stable Fisher estimation.
- Consider hybrid approaches (EWC + small replay buffer) for long task sequences.
- Evaluate using multiple metrics (AA, FM, BWT, Accuracy Stream) for a comprehensive assessment.

### 3.10.5 Key Takeaways for Reports

- EWC is a **regularization-based continual learning method** that leverages parameter importance.
- It is memory-efficient and theoretically principled but may underperform compared to replay-based approaches on large-scale, long sequences of tasks.
- Reporting results with metrics like Accuracy (Stream), FM, and BWT allows fair comparison with other methods such as naive fine-tuning and iCaRL.
- The method highlights the **stability–plasticity trade-off**, which is central to continual learning research.

**Conclusion :** EWC provides an elegant and practical solution to catastrophic forgetting, making it an essential baseline in class-incremental learning experiments. Its combination of theoretical grounding, low memory overhead, and ease of integration into standard training pipelines makes it suitable for both educational and research purposes.

# 4 High-Level Idea of iCaRL

## 4.1 Motivation

The failures of naive classifiers in class-incremental learning motivate the design of alternative strategies. An effective solution must :

- Reduce classifier bias toward new classes,
- Stabilize the feature representation across experiences,
- Operate under limited memory constraints.

iCaRL addresses these challenges by replacing the standard softmax classifier with an exemplar-based classification strategy and by explicitly preserving representative samples from past classes. These components are introduced and analyzed in the next sections.

## 4.2 Objective

The goal of iCaRL is to address catastrophic forgetting under these constraints. Specifically, iCaRL aims to :

- Learn new classes incrementally,
- Preserve performance on previously learned classes,
- Operate with a limited memory budget,
- Perform classification without task information at test time.

To achieve this, iCaRL combines representation learning, exemplar-based rehearsal, and a geometry-based classification strategy, which will be detailed in the following sections.

## 4.3 Rethinking Classification in Continual Learning

The key insight behind iCaRL is that catastrophic forgetting in class-incremental learning is not caused by a single factor, but by the interaction of two issues :

1. Instability of feature representations over time,
2. Bias of standard classifiers toward recently learned classes.

Rather than attempting to directly protect individual network parameters, as done in regularization-based methods (e.g., EWC), iCaRL adopts a different perspective. It focuses on preserving a stable *feature space* and performing classification based on the geometry of this space.

## 4.4 Core Principle of iCaRL

iCaRL is built upon the following principle :

> *If a neural network learns a discriminative and stable feature representation, then classification can be performed by comparing samples to representative feature prototypes of each class.*

Instead of relying solely on a softmax classifier trained incrementally, iCaRL maintains a small set of representative samples, called *exemplars*, for each class. These exemplars summarize the distribution of each class in the learned feature space.

## 4.5 Three Main Components of iCaRL

At a high level, iCaRL combines three tightly coupled components :

1. **Representation learning** : learning a feature extractor that produces meaningful embeddings for all classes,
2. **Exemplar memory** : storing a small number of representative samples per class,
3. **Nearest-Mean classification** : predicting labels based on distances to class means in feature space.

Each component addresses a specific weakness of naive incremental learning.

## 4.6 Learning a Stable Feature Space

The feature extractor $f_\theta(\cdot)$ maps each input image $x$ to a feature vector $f_\theta(x) \in R^d$. The goal is to ensure that :

— Samples from the same class cluster together,

— Samples from different classes are well separated,

— Feature representations of old classes remain stable as new classes are learned.

To prevent drastic changes in the feature space, iCaRL uses a form of knowledge distillation during training, ensuring that the responses of the network for old classes remain consistent across experiences.

## 4.7 Exemplar-Based Memory

Since storing all past data is infeasible, iCaRL maintains a memory buffer with a fixed capacity $M$. This memory is populated with a small number of exemplars per class.

Rather than selecting exemplars randomly, iCaRL carefully chooses samples that best represent the distribution of each class in feature space. These exemplars serve two purposes :

— They are replayed during training to reduce forgetting,

— They are used to compute class prototypes at inference time.

## 4.8 Classification by Geometry Instead of Logits

A key departure from standard neural network classifiers is that iCaRL does not rely on the softmax output for final predictions. Instead, classification is performed using a *Nearest-Mean Classifier* (NMC).

For each class $c$, a prototype (mean feature vector) is computed :

$$\mu_c = \frac{1}{|\mathcal{E}_c|} \sum_{x \in \mathcal{E}_c} f_\theta(x),$$

where $\mathcal{E}_c$ is the set of exemplars for class $c$.

Given a test sample $x$, the predicted label is :

$$\hat{y} = \arg\min_c \ \|f_\theta(x) - \mu_c\|_2.$$

This classification strategy is inherently less biased toward recent classes, as all classes are treated symmetrically through their prototypes.

### 4.9 Why This Design Works

By combining exemplar rehearsal, representation preservation, and distance-based classification, iCaRL achieves the following :

— Old classes remain represented through stored exemplars,

— Feature drift is reduced through distillation,

— Classifier bias is mitigated by avoiding softmax-based decisions.

As a result, iCaRL provides a principled solution to class-incremental learning that balances performance, memory efficiency, and scalability.

The following sections describe each component of iCaRL in detail, starting with the feature extractor and representation learning process.

## 5 Feature Extractor and Representation Learning in iCaRL

### 5.1 Role of the Feature Extractor

At the core of iCaRL lies a deep neural network used as a *feature extractor*. Rather than directly producing class probabilities, this network maps each input sample to a high-dimensional feature representation.

Formally, let :

$$f_\theta : \mathcal{X} \to R^d$$

denote the feature extractor parameterized by $\theta$, where $\mathcal{X}$ is the input space of images and $d$ is the feature dimensionality.

The output $f_\theta(x)$ is intended to capture semantic information about the input image $x$, such that samples from the same class are close in feature space, while samples from different classes are well separated.

### 5.2 Decoupling Representation and Classification

A fundamental design choice in iCaRL is the *decoupling* of representation learning and classification. In contrast to standard classifiers, where the final fully connected layer is trained to directly output class scores, iCaRL removes this dependency.

In practice, this is achieved by replacing the final classification layer with an identity mapping :

$$\texttt{fc} = \texttt{Identity}.$$

As a result, the network outputs feature vectors instead of logits. This design allows the feature extractor to be trained independently of the final classification mechanism.

### 5.3 Why a Stable Feature Space Is Crucial

In class-incremental learning, the feature space must satisfy stronger requirements than in standard supervised learning. Specifically, it must :

— Remain stable across learning experiences,

— Preserve the relative geometry of old classes,

— Accommodate new classes without collapsing previous ones.

If the feature representations of old classes drift significantly when new classes are learned, then any classifier built on top of those features becomes unreliable. Therefore, preserving the feature space is a primary objective of iCaRL.

## 5.4 Training the Feature Extractor Incrementally

During experience $t$, the feature extractor is trained using :

— Samples from the current experience $\mathcal{D}_t$,

— Stored exemplars from previous classes.

This mixture of new data and exemplars ensures that gradients are influenced by both old and new classes. As a result, updates to the parameters $\theta$ are constrained to preserve existing representations while learning new ones.

## 5.5 Distillation for Representation Preservation

To further reduce representation drift, iCaRL incorporates a distillation mechanism. Before training on experience $t$, a copy of the previous model $f_{\theta_{t-1}}$ is stored.

During training, the current model $f_{\theta_t}$ is encouraged to produce similar outputs to the previous model for samples of old classes. This is achieved by minimizing a distillation loss that penalizes deviations between old and new feature responses.

Intuitively, distillation acts as a regularizer that anchors the feature extractor to its previous behavior.

## 5.6 Illustrative Example

Consider a model trained on the first experience containing classes *airplane* and *automobile*. The feature extractor learns to separate these classes in feature space.

When the second experience introduces *bird* and *cat*, the feature extractor must learn new discriminative directions. However, thanks to exemplar replay and distillation, the learned feature clusters for airplane and automobile remain approximately in the same regions of the feature space.

This stability allows previously learned classes to remain identifiable even after multiple incremental updates.

## 5.7 Connection to Out Implementation

In our implementation, the feature extractor corresponds to the ResNet-18 backbone :

```
backbone = resnet18(pretrained=False)
backbone.fc = nn.Identity()
```

This explicitly enforces the separation between feature extraction and classification. The learned features are then used both for exemplar selection and for distance-based classification, as described in the next sections.

### 5.8 Summary

The feature extractor in iCaRL serves as a stable embedding function that maps images into a structured feature space. By decoupling representation learning from classification and by constraining updates through exemplars and distillation, iCaRL ensures that this feature space remains meaningful throughout the incremental learning process.

In the following section, we describe how iCaRL selects and stores representative samples from this feature space using an exemplar memory.

# 6 Exemplar Memory in iCaRL

## 6.1 Motivation for an Exemplar Memory

A central challenge in class-incremental learning is the inability to store and replay all previously seen data. However, completely discarding past data leads to catastrophic forgetting, as discussed in previous sections.

iCaRL addresses this problem by maintaining a small *exemplar memory*, which stores a limited number of representative samples from previously learned classes. This memory provides the model with a partial but informative view of past data, enabling it to preserve knowledge over time.

## 6.2 Memory Budget Constraint

Let $M$ denote the total memory budget, expressed as the maximum number of images that can be stored. This budget is fixed throughout training.

If the model has learned $K$ classes so far, the memory is divided equally among classes :

$$m = \left\lfloor \frac{M}{K} \right\rfloor$$

exemplars per class.

As new classes are introduced, the number of exemplars per class decreases to respect the fixed memory budget. This ensures a fair and balanced representation of all classes.

## 6.3 Class-Balanced Storage

Maintaining a balanced number of exemplars per class is critical. If some classes were overrepresented in memory, the model would become biased toward them during replay and classification.

By enforcing equal memory allocation :

— All classes contribute equally to rehearsal,

— No class dominates the feature space representation,

— Classification remains fair across experiences.

This design choice distinguishes iCaRL from naive replay methods that use random sampling.

## 6.4 What Is Stored in Memory

The exemplar memory stores :

— The raw input samples (e.g., images),

— Their corresponding class labels.

Importantly, feature representations are *not* stored. Instead, features are recomputed dynamically using the current feature extractor. This allows exemplars to adapt to gradual changes in the representation space.

## 6.5 Use of Exemplars During Training

During training on experience $t$, the model is trained on :

— New samples from $\mathcal{D}_t$,

— Exemplars from all previously learned classes.

This mixed training set ensures that gradient updates reflect both old and new knowledge. As a result :

— Feature representations of old classes are reinforced,

— The classifier does not become overly biased toward new classes.

## 6.6 Illustrative Example

Consider a total memory budget of $M = 2000$ images. After learning the first two classes, the memory stores 1000 exemplars per class.

After learning four classes, the memory is rebalanced to store 500 exemplars per class. Previously stored exemplars are reduced accordingly, preserving only the most representative samples.

By the end of training on CIFAR-10 (10 classes), the memory contains 200 exemplars per class.

## 6.7 Why a Small Memory Is Sufficient

Although the exemplar memory contains only a small fraction of the full dataset, it is surprisingly effective. This is because :

— Exemplars are selected to approximate the class distribution in feature space,

— Redundant samples are avoided,

— Classification relies on class means rather than individual samples.

Thus, even a limited memory budget can preserve the essential structure of the data.

## 6.8 Connection to Our Implementation

In our implementation, the exemplar memory is controlled by the parameter :

```
memory_size = 2000
```

This value specifies the total number of stored exemplars across all classes. The class-balanced allocation and memory updates are handled internally by the iCaRL strategy in Avalanche.

## 6.9 Summary

The exemplar memory is a cornerstone of iCaRL. By storing a carefully selected and balanced subset of past data, it enables rehearsal, stabilizes feature learning, and supports unbiased classification. In the next section, we explain how these exemplars are selected using the herding procedure.

# 7 Herding : Selecting Exemplars in iCaRL

## 7.1 Motivation

Simply storing random samples from each class is often insufficient. Random selection may :

— Include outliers or atypical samples,

— Miss key regions of the feature space,

— Fail to accurately represent the class mean.

To overcome this, iCaRL employs a procedure called *herding*, which selects exemplars that best approximate the class distribution in feature space.

## 7.2 Class Mean in Feature Space

Let $\mathcal{D}_c$ be the set of all training samples of class $c$ in the current experience. The mean feature vector of class $c$ is defined as :

$$\mu_c = \frac{1}{|\mathcal{D}_c|} \sum_{x \in \mathcal{D}_c} f_\theta(x),$$

where $f_\theta(x)$ is the feature representation of $x$.

This mean vector serves as a *target* for selecting exemplars, ensuring that the chosen samples collectively approximate the class center.

## 7.3 Herding Procedure

Suppose we want to select $m$ exemplars for class $c$. Herding proceeds iteratively :

1. Initialize the set of selected exemplars $\mathcal{E}_c = \emptyset$.
2. For $k = 1$ to $m$ :
   — For each candidate $x \in \mathcal{D}_c \setminus \mathcal{E}_c$, compute the temporary mean if $x$ is added :

$$\bar{\mu}_k = \frac{1}{k} \left( f_\theta(x) + \sum_{x' \in \mathcal{E}_c} f_\theta(x') \right)$$

   — Select the sample $x^*$ that minimizes the distance to the true class mean :

$$x^* = \arg\min_x \|\mu_c - \bar{\mu}_k\|_2$$

   — Add $x^*$ to $\mathcal{E}_c$

At the end of this process, $\mathcal{E}_c$ contains $m$ exemplars whose mean closely matches the class mean $\mu_c$.

### 7.4 Intuition Behind Herding

Herding ensures that :

— The selected exemplars are *representative* of the class in feature space,

— Extreme or outlier samples are avoided,

— The average feature vector of the exemplars approximates the true class center.

This is critical because iCaRL uses the class mean of exemplars for nearest-mean classification at test time.

### 7.5 Illustrative Example

Consider a class $c$ with feature vectors for 5 images :

$$f_\theta(x_1) = [0.2, 0.3], f_\theta(x_2) = [0.4, 0.5], f_\theta(x_3) = [0.1, 0.2], f_\theta(x_4) = [0.3, 0.4], f_\theta(x_5) = [0.8, 0.9]$$

The class mean is :

$$\mu_c = \frac{1}{5}\sum_{i=1}^{5} f_\theta(x_i) = [0.36, 0.46]$$

If we want $m = 3$ exemplars, herding will iteratively select the samples that keep the running mean as close as possible to $\mu_c$, likely avoiding the outlier $x_5 = [0.8, 0.9]$ until necessary.

### 7.6 Connection to Implementation

In Avalanche implementation, this process is handled internally by the 'ICaRL' strategy. We specify the memory size, and herding automatically selects exemplars from each class as new experiences arrive.

### 7.7 Summary

Herding ensures that even a small exemplar memory can faithfully represent all classes in feature space. This careful selection is what allows iCaRL to :

— Preserve knowledge of old classes with limited memory,

— Perform accurate nearest-mean classification,

— Mitigate catastrophic forgetting effectively.

In the next section, we will examine how the model is trained using a combination of classification and distillation losses to further preserve knowledge.

# 8 Training Loss in iCaRL : Classification and Distillation

### 8.1 Overview

During each incremental experience $t$, iCaRL must train the model to :

1. Learn to classify new classes in the current dataset $\mathcal{D}_t$,

2. Preserve knowledge about old classes stored in the exemplar memory $\mathcal{M}$.

To achieve this, iCaRL combines two loss components :

— **Classification loss** for new classes,

— **Distillation loss** for old classes.

This combination allows the network to adapt to new information while constraining updates to avoid catastrophic forgetting.

## 8.2 Classification Loss for New Classes

For samples $(x_i, y_i) \in \mathcal{D}_t$, the standard cross-entropy loss is used :

$$\mathcal{L}_{CE} = -\frac{1}{|\mathcal{D}_t|}\sum_{i=1}^{|\mathcal{D}_t|} \log p_\theta(y_i \mid x_i),$$

where $p_\theta(y_i \mid x_i)$ is the softmax probability of class $y_i$ computed using the classifier (for new classes only).

This loss encourages the network to produce high scores for the new classes in the current experience.

## 8.3 Distillation Loss for Old Classes

To preserve old knowledge, iCaRL applies a *distillation loss* on exemplars of previously learned classes :

$$\mathcal{L}_{distill} = -\frac{1}{|\mathcal{M}|}\sum_{(x_j,y_j)\in\mathcal{M}}\sum_{c=1}^{K_{t-1}} q_c^{(j)} \log p_c^{(j)}$$

where :

— $K_{t-1}$ is the number of old classes,

— $q_c^{(j)}$ is the output probability of the old model for class $c$ on sample $x_j$,

— $p_c^{(j)}$ is the current model's predicted probability for class $c$,

— $\mathcal{M}$ is the set of exemplars from old classes.

This loss forces the current network to mimic the behavior of the previous network on old classes, anchoring its predictions and reducing feature drift.

## 8.4 Total Loss

The total training loss during experience $t$ is :

$$\mathcal{L}_{total} = \mathcal{L}_{CE} + \mathcal{L}_{distill}$$

This simple additive combination balances learning new information with preserving old knowledge.

### 8.5 Intuition Behind Distillation

Distillation can be interpreted as a form of soft regularization :

— The old model's outputs $q_c^{(j)}$ contain "soft targets" for old classes,

— The network is guided to reproduce these outputs instead of blindly fitting new data,

— This prevents the old class representations from drifting too far in feature space.

In essence, distillation replaces direct access to old data with a *behavioral snapshot* of the network.

### 8.6 Illustrative Example

Suppose the model has learned classes *airplane* and *automobile*, and is now learning *bird* and *cat*.

1. New samples from bird and cat contribute to $\mathcal{L}_{CE}$, training the network to recognize these new classes.
2. Stored exemplars of airplane and automobile are used in $\mathcal{L}_{distill}$, guiding the network to maintain its previous behavior on these classes.

By combining these two losses, the network learns new classes without completely overwriting old representations.

### 8.7 Connection to Implementation

In your Avalanche implementation, the loss combination is handled internally by the `ICaRL` strategy. You provide the optimizer, feature extractor, and classifier, and Avalanche automatically computes classification and distillation losses during training.

### 8.8 Summary

The training loss in iCaRL ensures incremental learning works effectively :

— **Classification loss** teaches the model to recognize new classes,

— **Distillation loss** preserves old knowledge,

— The combination minimizes catastrophic forgetting while allowing the feature space to adapt.

In the next section, we will discuss how iCaRL performs inference using the nearest-mean classifier, completing the learning pipeline.

# 9 Inference with Nearest-Mean Classifier in iCaRL

### 9.1 Why Standard Inference Fails

In standard neural network classifiers, inference is performed by selecting the class with the highest softmax score. However, as discussed earlier, softmax-based inference is unreliable in class-incremental learning due to :

— Classifier bias toward recently learned classes,

— Poor calibration of logits for old classes,

— Mismatch between training and evaluation distributions.

To overcome these limitations, iCaRL replaces softmax-based inference with a geometry-based classification strategy.

## 9.2 Nearest-Mean Classifier (NMC)

Instead of relying on classifier logits, iCaRL performs inference in the learned feature space using a *Nearest-Mean Classifier*.

For each class $c$, a prototype vector (class mean) is computed from the exemplars :

$$\mu_c = \frac{1}{|\mathcal{E}_c|} \sum_{x \in \mathcal{E}_c} f_\theta(x),$$

where :

— $\mathcal{E}_c$ is the exemplar set of class $c$,

— $f_\theta(x)$ is the feature representation of sample $x$.

## 9.3 Prediction Rule

Given a test sample $x$, its feature representation is computed using the current feature extractor :

$$z = f_\theta(x).$$

The predicted class label is obtained by minimizing the Euclidean distance between $z$ and the class means :

$$\hat{y} = \arg\min_c \; \|z - \mu_c\|_2.$$

This rule assigns the test sample to the class whose exemplar mean is closest in feature space.

## 9.4 Intuition Behind Nearest-Mean Classification

The Nearest-Mean Classifier relies on the assumption that samples of the same class form compact clusters in feature space. By representing each class with a single prototype vector, iCaRL :

— Treats all classes symmetrically,

— Avoids bias introduced by uneven training,

— Leverages the geometry of the learned representation.

Unlike softmax classifiers, NMC does not depend on classifier weights that may become biased during incremental updates.

## 9.5 Illustrative Example

Consider three classes with the following exemplar means in a two-dimensional feature space :

$$\mu_1 = [0.2, 0.3], \quad \mu_2 = [0.6, 0.5], \quad \mu_3 = [0.4, 0.8].$$

For a test sample with feature vector :

$$z = [0.45, 0.52],$$

the predicted class is :

$$\hat{y} = \arg\min_c \|z - \mu_c\|_2 = 2.$$

Thus, the sample is classified as belonging to class 2, based solely on proximity in feature space.

## 9.6 Advantages of Nearest-Mean Inference

Nearest-mean inference offers several advantages in class-incremental learning :

— Robustness to class imbalance,

— Reduced sensitivity to representation drift,

— Compatibility with fixed memory constraints,

— Simplicity and computational efficiency.

These properties make it particularly well-suited for continual learning scenarios.

## 9.7 Connection to Implementation

In your Avalanche implementation, nearest-mean inference is automatically handled by the `ICaRL` strategy. Even though a classifier layer is trained during learning, it is not used during evaluation. Instead, predictions are made using the exemplar means computed from the stored memory.

## 9.8 Summary

Inference in iCaRL is performed using a nearest-mean classifier in the learned feature space. By avoiding softmax-based predictions and relying on exemplar geometry, iCaRL ensures fair and stable classification across all classes, even after multiple incremental learning experiences.

In the next section, we will bring all components together and describe a full incremental learning cycle of iCaRL from start to finish.

# 10 Putting It All Together : One Full iCaRL Learning Cycle

## 10.1 Overview of an Incremental Learning Step

We now describe how all components of iCaRL interact during a single incremental learning experience. This section summarizes the complete workflow of iCaRL, from receiving new data to performing inference after training.

At experience $t$, the model has already learned $K_{t-1}$ classes and maintains :

— A feature extractor $f_{\theta_{t-1}}$,

— An exemplar memory $\mathcal{M}_{t-1}$,

— A set of class means $\{\mu_1, \ldots, \mu_{K_{t-1}}\}$.

The goal is to incorporate new classes $\mathcal{C}_t$ while preserving performance on old classes.

## 10.2 Step-by-Step Learning Procedure

**Step 1 : Receive New Experience** The model receives a new dataset $\mathcal{D}_t$ containing samples from previously unseen classes $\mathcal{C}_t$. No samples from old classes are available, except those stored in the exemplar memory.

**Step 2 : Store Previous Model** Before training begins, a copy of the current model $f_{\theta_{t-1}}$ is stored. This frozen model will be used to compute distillation targets for old classes.

**Step 3 : Construct the Training Set** The training set is formed by combining :

— New samples from $\mathcal{D}_t$,

— Exemplars from the memory $\mathcal{M}_{t-1}$.

This mixed dataset ensures that training gradients reflect both old and new knowledge.

**Step 4 : Train the Feature Extractor** The model is trained by minimizing the total loss :

$$\mathcal{L}_{total} = \mathcal{L}_{CE} + \mathcal{L}_{distill},$$

where :

— $\mathcal{L}_{CE}$ learns the new classes,

— $\mathcal{L}_{distill}$ preserves old class behavior.

This step updates the parameters $\theta_t$ of the feature extractor.

**Step 5 : Update the Exemplar Memory** Once training is completed, the exemplar memory is updated :

— The total number of classes is now $K_t = K_{t-1} + |\mathcal{C}_t|$,

— The memory budget $M$ is divided equally across classes,

— Existing exemplar sets are reduced if necessary,

— New exemplars are selected for each new class using herding.

The updated memory is denoted $\mathcal{M}_t$.

**Step 6 : Compute Class Means** For each class $c$, the class mean is recomputed using the updated exemplar set :

$$\mu_c = \frac{1}{|\mathcal{E}_c|} \sum_{x \in \mathcal{E}_c} f_{\theta_t}(x).$$

These means are stored for use during inference.

### 10.3 Inference After Experience $t$

At test time, given a sample $x$, the model :

1. Computes its feature representation $z = f_{\theta_t}(x)$,
2. Compares $z$ to all class means $\{\mu_1, \dots, \mu_{K_t}\}$,
3. Assigns the label of the nearest mean :

$$\hat{y} = \arg\min_c \|z - \mu_c\|_2.$$

Importantly, this inference procedure does not require task identity and treats all classes equally.

### 10.4 Algorithmic Summary

The complete iCaRL procedure is summarized in Algorithm 2.

**Algorithm 2** iCaRL Incremental Learning Algorithm

```
1: Initialize feature extractor $f_{\theta_0}$
2: Initialize empty memory $\mathcal{M}_0$
3: for each experience $t = 1, \dots, T$ do
4:     Receive dataset $\mathcal{D}_t$
5:     Store previous model $f_{\theta_{t-1}}$
6:     Train on $\mathcal{D}_t \cup \mathcal{M}_{t-1}$ using $\mathcal{L}_{CE} + \mathcal{L}_{distill}$
7:     Update exemplar memory using herding
8:     Recompute class means
9: end for
10: Perform inference using nearest-mean classification
```

### 10.5 Summary

This section presented a complete view of the iCaRL method as a sequence of well-defined steps. Each component—representation learning, exemplar memory, herding, distillation, and nearest-mean inference—plays a critical role in enabling effective class-incremental learning.

In the final section, we analyze why iCaRL works well in practice and discuss its limitations.

## 11 Why iCaRL Works : Strengths, Limitations, and Practical Considerations

This final section analyzes the behavior of iCaRL from a conceptual and practical perspective. We explain why the method is effective in class-incremental learning, identify its limitations, and position it relative to other continual learning strategies.

## 11.1 Why iCaRL Mitigates Catastrophic Forgetting

Catastrophic forgetting occurs when learning new tasks overwrites representations useful for old tasks. iCaRL addresses this issue through the combination of three complementary mechanisms.

### 11.1.1 Exemplar-Based Rehearsal

Unlike regularization-based methods (e.g., EWC), iCaRL explicitly stores a subset of old samples. These exemplars :

— Reintroduce old-class gradients during training,

— Anchor the feature space to previously learned distributions,

— Prevent representation drift.

Because exemplars are real data points, rehearsal provides a strong and stable signal compared to purely constraint-based methods.

### 11.1.2 Knowledge Distillation for Old Classes

The distillation loss preserves the output behavior of the previous model :

$$\mathcal{L}_{distill} = -\sum_{c=1}^{K_{t-1}} q_c(x) \log p_c(x),$$

where $q_c(x)$ are soft targets produced by the frozen old model.

This prevents the classifier from reallocating probability mass away from old classes, even when no full old dataset is available.

### 11.1.3 Nearest-Mean Classification

By using class prototypes instead of a learned classifier head, iCaRL :

— Avoids classifier bias toward recently learned classes,

— Maintains a balanced decision boundary,

— Decouples representation learning from classification.

This design choice is particularly important in class-incremental scenarios, where class imbalance is unavoidable.

## 11.2 Comparison with Other Continual Learning Strategies

### 11.2.1 iCaRL vs. Naive Fine-Tuning

Naive fine-tuning trains only on new data and therefore :

— Quickly forgets old classes,

— Produces misleadingly high short-term accuracy,

— Is unsuitable for continual learning.

In contrast, iCaRL explicitly preserves old knowledge through memory and distillation.

### 11.2.2 iCaRL vs. EWC

Elastic Weight Consolidation constrains important parameters using a quadratic penalty :

$$\mathcal{L}_{EWC} = \sum_i \frac{\lambda}{2} F_i (\theta_i - \theta_i^*)^2.$$

While EWC is memory-efficient, it :

— Struggles with long task sequences,

— Accumulates approximation errors,

— Cannot recover forgotten information.

iCaRL, although memory-based, scales better in terms of retained accuracy.

### 11.2.3 iCaRL vs. Other Replay-Based Methods

Compared to random replay or reservoir sampling, iCaRL :

— Selects exemplars strategically via herding,

— Uses representation-aware memory,

— Combines replay with distillation.

This makes iCaRL more sample-efficient and stable.

## 11.3 Limitations of iCaRL

Despite its strengths, iCaRL has several limitations.

### 11.3.1 Memory Requirement

The method requires storing raw images :

— This may violate privacy constraints,

— Memory size limits performance,

— Scalability becomes challenging for large datasets.

### 11.3.2 Computational Cost

Each experience involves :

— Recomputing class means,

— Re-selecting exemplars,

— Training on an increasing number of classes.

This leads to longer training times compared to regularization-based methods.

### 11.3.3 Prototype Approximation Error

Class means are only approximations of class distributions. If the feature space is poorly learned, nearest-mean classification may become suboptimal.

## 11.4 Practical Guidelines

From empirical practice, the following recommendations emerge :

— Use iCaRL when memory storage is acceptable,

— Fix the memory budget when comparing with other replay-based methods,

— Evaluate using stream accuracy and forgetting metrics,

— Prefer SGD with momentum for stable representation learning.

## 11.5 Conclusion

iCaRL represents a principled and effective approach to class-incremental learning. By combining exemplar rehearsal, knowledge distillation, and prototype-based inference, it achieves a strong balance between plasticity and stability.

Although memory-based, iCaRL remains a foundational method that continues to inspire modern continual learning techniques.