UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

DIPARTIMENTO DI SCIENZE FISICHE, INFORMATICHE E MATEMATICHE

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

STAER: A Temporal Aligned Rehearsal Spiking Neural Network for Continual Learning

Candidato:

Matteo Gianferrari

Relatore:

Prof. Simone Calderara

Correlatori:

Dott. Riccardo Salami

Dott.sa Omayma Moussadek

Dott. Cosimo Fiorini

ANNO ACCADEMICO 2024-2025

Sommario

Il Continual Learning (CL) affronta il problema dell'apprendimento da flussi di dati sequenziali, nei quali un modello deve acquisire nuove conoscenze senza compromettere quelle apprese in precedenza. In questo contesto, una delle difficoltà principali è la catastrophic forgetting, ossia la tendenza delle reti neurali a sovrascrivere le rappresentazioni acquisite quando vengono addestrate su nuovi task. Sebbene molte strategie di replay abbiano mostrato risultati efficaci nelle reti neurali artificiali tradizionali, la loro applicazione alle Spiking Neural Networks (SNNs) presenta sfide aggiuntive, poiché in questi modelli l'informazione non è rappresentata solo nello spazio delle attivazioni, ma anche nella dinamica temporale delle risposte neurali.

Questa tesi propone STAER (Spiking Temporal Alignment with Experience Replay), un metodo di continual learning per SNNs che combina experience replay e allineamento temporale delle traiettorie dei logit. L'idea centrale è che, per preservare la conoscenza passata, non sia sufficiente mantenere corrette le predizioni finali, ma sia necessario anche conservare una dinamica temporale compatibile con quella appresa in precedenza. A tal fine, il metodo memorizza nel buffer tracce temporali a più scale e introduce un obiettivo di allineamento basato su Soft-DTW divergence, applicato a

versioni compresse, originali ed espanse della risposta del modello.

La valutazione sperimentale, condotta sui benchmark Sequential-MNIST e Sequential-CIFAR10 nei protocolli Task-Incremental e Class-Incremental, mostra che STAER migliora in modo consistente rispetto alle principali baseline di replay in ambito spiking, riducendo il forgetting e aumentando l'accuracy finale, in particolare nello scenario Class-Incremental. I risultati indicano che modellare esplicitamente la dimensione temporale costituisce un fattore chiave per rendere le SNNs più efficaci e competitive nel continual learning.

Abstract

Continual Learning (CL) addresses the problem of learning from sequential data streams, where a model must acquire new knowledge without degrading what it has previously learned. In this setting, one of the main challenges is catastrophic forgetting, namely the tendency of neural networks to overwrite past representations when trained on new tasks. While replay-based strategies have proven effective in conventional Artificial Neural Networks, their application to Spiking Neural Networks (SNNs) introduces additional difficulties, since information in these models is encoded not only in activation values but also in the temporal dynamics of neural responses.

This thesis proposes STAER (Spiking Temporal Alignment with Experience Replay), a continual learning method for SNNs that combines experience replay with temporal alignment of logit trajectories. The central idea is that preserving past knowledge in a spiking model should not be limited to maintaining correct final predictions, but should also enforce temporal consistency in the evolution of the response. To this end, the method stores multi-scale temporal traces in the replay buffer and introduces an alignment objective based on the Soft-DTW divergence, applied to compressed, original, and expanded versions of the model response.

The experimental evaluation, conducted on Sequential-MNIST and Sequential-CIFAR10 under both Task-Incremental and Class-Incremental protocols, shows that STAER consistently improves over the main replay-based spiking baselines, reducing forgetting and increasing final accuracy, especially in the more challenging Class-Incremental setting. Overall, the results suggest that explicitly modeling the temporal dimension is a key ingredient for making SNNs more effective and competitive in continual learning.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Artificial Intelligence (**AI**) and Machine Learning (**ML**) have become central tools for building systems that can extract patterns from data and support complex decision-making. In particular, Deep Learning (**DL**) has achieved strong results in several domains by learning task-relevant representations directly from the data. However, most modern models are still trained in an *offline setting*: they assume access to a fixed dataset and are optimized for a static task distribution. This assumption is often unrealistic in real-world applications, where data arrive *sequentially* and learning systems should *adapt over time*.

A key limitation of standard neural networks is their difficulty in continuously learning without degrading previously acquired knowledge. When trained on new data or tasks, models tend to overwrite past representations, a phenomenon known as *catastrophic forgetting*. Continual Learning (**CL**) addresses this problem by studying methods that allow a model to incorporate new knowledge while preserving perfor-

mance on previously learned tasks. The core challenge of **CL** is to balance *plasticity* (learning new information) and *stability* (retaining prior knowledge).

Among the different Continual Learning settings, Class-Incremental Learning (Class-IL) is particularly challenging and practically relevant. In this scenario, new classes are introduced over time, but the model must perform inference over all classes seen so far without access to the task identity at test time. This makes the problem significantly harder than task-aware settings, since the system must maintain discriminative representations as the label space expands. In realistic deployments, this challenge is further amplified by memory constraints, non-stationary data streams, and the absence of explicit task boundaries.

In parallel with the development of **CL**, there is growing interest in neural models that are more aligned with *biological computation*. Spiking Neural Networks (**SNNs**) are a prominent example: unlike conventional Artificial Neural Networks (**ANNs**), **SNNs** process information through *discrete spikes* and explicitly model *temporal dynamics*. Their event-driven nature and temporal processing capabilities make them a promising framework for online Continual Learning, especially in settings where timing carries meaningful information. More broadly, **SNNs** provide a natural ground for exploring biologically inspired principles such as *temporal coding* and *local neural dynamics*.

Despite their potential, the effective application of **SNNs** in Continual Learning remains an open challenge. Although *replay-based* and *regularization-based* strategies have been widely studied in **ANN**-based **CL**, these methods do not directly account for a key aspect of **SNNs**: the *temporal structure* of neural responses. In spiking mod-

els, forgetting may not only appear as a drop in classification accuracy, but also as a degradation or shift in spike timing patterns learned for previous classes. Therefore, preserving *temporal consistency* becomes an important requirement for Continual Learning in **SNNs**.

This thesis addresses this problem by proposing a novel **CL** method for **SNNs** that integrates *temporal dilation* and *compression* into a *replay-based* strategy. In this work we propose **S**piking **T**emporal **A**lignment with **E**xperience **R**eplay (**STAER**) [2], and is motivated by the idea that past knowledge in **SNNs** should be preserved not only at the level of outputs, but also at the level of their *temporal evolution*. In particular, the method combines replay with a *temporal alignment objective* and multi-scale temporal transformations (*dilation*/*compression*) applied to replayed responses, in order to improve robustness to temporal drift during continual updates.

The central hypothesis of this thesis is that *time-aware* replay improves knowledge retention in Spiking Continual Learning. Standard replay can help preserve class information, but it may be insufficient when the internal temporal dynamics of the model change across incremental training phases. By explicitly constraining the temporal structure of replayed responses—and by exposing the model to *compressed and dilated temporal variants*—the proposed method encourages more stable spike-based representations across tasks. This design is both methodologically useful for Class-IL performance and conceptually consistent with the temporal nature of spiking computation.

The research developed in this thesis has also led to the publication of a paper presenting **STAER** [2], currently available as an arXiv preprint and under review for an international conference in the field of Artificial Intelligence. For this reason, several figures and tables reported in the present manuscript are directly reproduced from that publication. When not explicitly indicated otherwise, these materials should be considered as originating from [2].

The remainder of this thesis is organized as follows. Chapter 2 introduces the background on Machine Learning, Deep Learning, Continual Learning, and the foundations of Spiking Neural Networks. Chapter 3 presents methods and principles most relevant to this work. Chapter 4 describes the proposed method in detail, including the temporal alignment objective and the dilation/compression mechanism. Chapter 5 reports the experimental setup, results, and ablation analysis. Finally, Chapter 6 concludes the thesis by summarizing the main findings and outlining future research directions.

Chapter 2

Machine Learning and Neural Networks

This chapter provides the theoretical background underlying the methods discussed in this thesis. It begins with the fundamental principles of Machine Learning and then introduces the main concepts of Deep Neural Networks, including their architectures and training mechanisms. The chapter finally addresses Continual Learning and Spiking Neural Networks, which constitute the broader context of the work presented in the following chapters.

2.1 Machine Learning

Machine Learning is the study of models that improve their performance on a task through experience. In the *supervised* setting, this experience is provided by a dataset of labeled examples

$$D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N},$$

where $\mathbf{x}_i \in X$ denotes the input and $y_i \in Y$ the corresponding target. The goal is to learn a function $f_\theta : X \to Y$, parameterized by θ, that generalizes well to unseen samples drawn from the same underlying distribution. Training is therefore not limited to fitting the observed data, but aims at extracting regularities that remain valid beyond the training set.

In a standard supervised classification problem, the learning objective is commonly formulated through *Empirical Risk Minimization* (**ERM**):

$$\theta^* = \arg\min_\theta \frac{1}{N} \sum_{i=1}^{N} \ell\big(y_i, f_\theta(\mathbf{x}_i)\big), \tag{2.1}$$

where $\ell(\cdot)$ is a loss function that measures the discrepancy between the predicted output and the ground truth. This formulation assumes that training and test data are sampled according to a compatible distribution, often summarized by the *i.i.d.* assumption (independent and identically distributed samples).

Beyond supervised learning, other paradigms are possible. In *unsupervised learning*, the dataset is unlabeled and the objective is to discover structure in the data, for instance through clustering, density estimation, or representation learning. In *self-supervised* and *semi-supervised* settings, only part of the supervision is explicit, and the model exploits either intrinsic structure or a limited amount of labels. In this thesis, however, the primary focus is on supervised classification, both in the conventional setting and in its continual counterpart.

2.1.1 Overfitting and Underfitting

A central objective in **ML** is *generalization*, namely the ability of a model to maintain good performance on unseen data. Two opposite failure modes are typically considered:

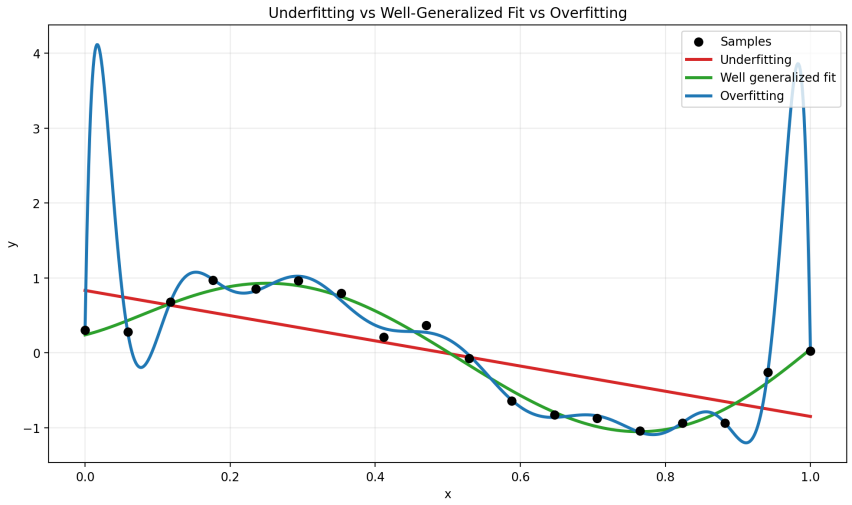

Figure 2.1: Visual representation of underfitting, overfitting, and a well-generalized curve fitting.

underfitting and *overfitting*.

Underfitting occurs when the model is not expressive enough, or is not trained sufficiently, to capture the relevant structure of the data. In this case, both training and test performance remain poor, since the model fails to approximate the target function even on the observed samples.

Overfitting, by contrast, occurs when the model adapts excessively to the training set, including noise, or dataset-specific artifacts that do not reflect the underlying data-generating process. In this regime, training performance can become very high while test performance deteriorates. Overfitting is particularly likely when the representational capacity of the model is large relative to the amount of available data, or when training is prolonged without appropriate *regularization*.

The trade-off between these two regimes is closely related to model selection. Increasing the capacity of a model can reduce underfitting, but may also increase the

risk of overfitting. The role of optimization, regularization, and validation protocols is therefore to identify a regime in which the model is sufficiently expressive to learn the task, yet constrained enough to preserve *generalization*. Figure 2.1 provides a schematic illustration of these three regimes in a simple setting.

2.1.2 Loss Function

The *loss function* quantifies the error made by the model on a sample or on a batch of samples, and provides the learning signal used during optimization. In supervised learning, given a prediction $\hat{y} = f_\theta(\mathbf{x})$ and a target y, the loss assigns a non-negative scalar value

$$\ell(y, \hat{y}) \geq 0,$$

which is small when the prediction is correct and large when it is inaccurate. The optimization process then aims to minimize the average loss over the training set.

In *multiclass classification*, the most common choice is the *cross-entropy loss*. Let $\mathbf{z} \in \mathbb{R}^K$ be the vector of logits produced by the model for K classes. The corresponding probability vector $\mathbf{p} \in [0,1]^K$ is obtained through the *softmax* function:

$$p_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \qquad k = 1, \ldots, K. \tag{2.2}$$

If the target label is represented by a one-hot vector $\mathbf{y} \in \{0,1\}^K$, the cross-entropy for a single sample is

$$\ell_{\mathrm{CE}}(\mathbf{y}, \mathbf{p}) = -\sum_{k=1}^{K} y_k \log p_k. \tag{2.3}$$

Since only the component corresponding to the correct class is equal to 1, the loss penalizes the model when it assigns low probability to the true label.

The choice of the loss function depends on the task formulation. For *regression*, one often adopts the Mean Squared Error (MSE); for binary classification, Binary Cross-Entropy is typically used. In all cases, the loss plays a dual role: it defines the training objective and determines the gradients used to update the model parameters.

2.1.3 Regularization

Machine Learning and Deep Learning models can possess a high representational capacity, which makes them effective approximators, but also exposes them to overfitting. *Regularization* denotes the set of techniques used to constrain learning so as to improve generalization.

A first class of methods acts directly on the model parameters or on the forward computation:

- ℓ_2 **regularization**. Also known as *weight decay*, it penalizes large parameter values by adding a quadratic term to the objective. This generally encourages smoother solutions and improves numerical stability.

- ℓ_1 **regularization**. It penalizes the absolute value of the parameters, often promoting sparse solutions and, implicitly, feature selection.

- **Batch Normalization (BN)**. Although primarily introduced to stabilize optimization, it can also have a regularizing effect because normalization statistics are estimated on mini-batches, thus introducing noise during training.

- **Dropout**. During training, a random subset of activations is set to zero. This prevents an excessive co-adaptation of neurons and can be interpreted as an implicit ensemble of sub-networks.

A second class of methods acts on the training protocol or on the data rather than on the architecture itself:

- **Early stopping**. Training is interrupted when the validation performance stops improving, preventing the optimizer from adapting too closely to the training set.

- **Data augmentation**. Additional training samples are generated by applying label-preserving transformations to the original data (e.g., crops, flips, rotations, or color perturbations in image classification). This increases dataset diversity and encourages invariance to irrelevant transformations.

In practice, strong performance is rarely obtained through a single regularization mechanism. Rather, modern training pipelines usually combine multiple techniques in order to balance optimization efficiency, representational capacity, and generalization.

2.1.4 Gradient Descent

Once the loss function has been defined, model parameters are learned through iterative optimization. The most common family of methods is based on *gradient descent*, which updates the parameters in the opposite direction of the gradient of the objective. Given a parameter vector θ and learning rate $\eta > 0$, the basic update rule is

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta \mathscr{L}(\theta^{(t)}), \tag{2.4}$$

where $\mathscr{L}(\theta)$ denotes the objective function evaluated at iteration t.

In large-scale learning, computing the exact gradient over the entire dataset at each step is usually too expensive. For this reason, training is commonly performed on *mini-batches*. If $\mathscr{B}$ denotes a subset of the training set, the update becomes

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta \mathscr{L}_{\mathscr{B}}(\theta^{(t)}). \tag{2.5}$$

This approximation introduces noise in the optimization process, but significantly reduces the computational cost and often improves generalization. The special case in which the mini-batch contains a single sample is referred to as *Stochastic Gradient Descent* (**SGD**).

The learning rate is a crucial hyperparameter. If it is too small, convergence may be excessively slow; if it is too large, the optimization may oscillate or diverge. In practice, gradient descent is often combined with momentum terms, adaptive learning-rate schedules, or more advanced optimizers, but the underlying principle remains unchanged: parameters are iteratively adjusted to reduce the training loss.

2.2 Deep Learning

The study of Artificial Neural Networks can be traced to early computational models of biological neurons proposed in the 1940s, which later inspired the *Perceptron*, a single-layer linear classifier. Subsequent progress led to the development of Multilayer Perceptrons (**MLPs**), which strongly increased the representational capacity by stacking multiple non-linear transformations. Despite this conceptual advance, research and practical adoption were historically constrained by limited computational resources

and restricted access to large-scale training data. In recent years, the combination of improved hardware acceleration and the availability of large datasets has driven a renewed interest in Deep Learning.

A key characteristic of neural network-based models is that their achievable performance is strongly influenced by *scale*, including the amount of training data, model capacity (e.g., *depth* and *width*), and available computation. As these factors increase, performance often continues to improve, provided that optimization and regularization techniques are appropriately managed. By contrast, many classical **ML** methods (e.g., *Logistic Regression* and *Support Vector Machines*) exhibit diminishing returns beyond a certain point: once the representational capacity of the chosen feature space is saturated, additional data may yield only marginal gains.

This qualitative difference is illustrated in Figure 2.2, where Deep Learning performance continues to increase with additional data, whereas classical machine learning tends to approach a plateau. The *plateauing behavior* is commonly associated with a fixed, human-designed, or manually selected representation, which can limit further improvements even when more training examples are available.

Traditional Machine Learning pipelines are typically organized into two stages:

1. **Pre-processing and feature engineering**: converting raw inputs into a set of informative features through automated procedures and/or domain knowledge (e.g., Principal Component Analysis (**PCA**) and task-specific descriptors);

2. **Model training**: learning a predictive model (or an ensemble) on the engineered feature space to optimize performance under a chosen objective.

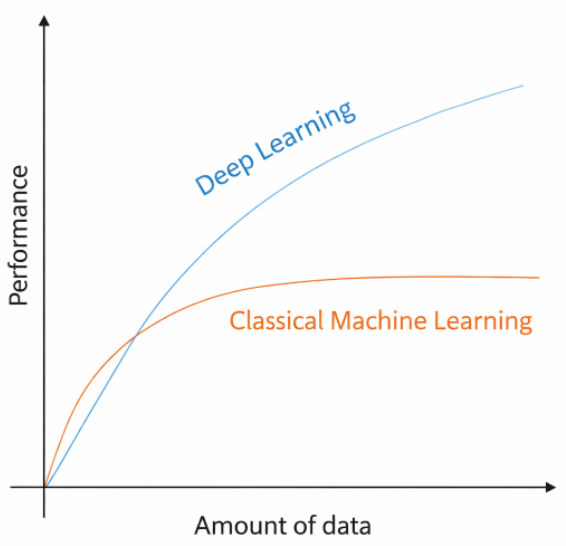

Figure 2.2: Qualitative comparison of performance trends as a function of data scale: deep learning models often continue to improve with more data, while classical machine learning methods may reach a performance plateau due to limited representational capacity and reliance on fixed features.

By contrast, Deep Learning methods aim to learn representations relevant to the task directly from the data. Instead of relying on a fixed, manually constructed feature space, the network progressively transforms input through multiple *layers* and *non-linear activations*, yielding hierarchical representations suited to the learning objective. With sufficient model capacity and appropriate regularization, increasing the quantity and diversity of training data generally improves the quality of these representations, thereby enhancing generalization and enabling a closer approximation to the underlying target mapping.

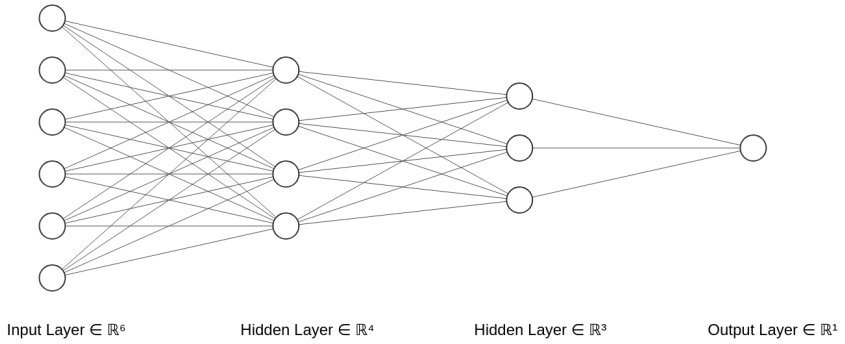

Figure 2.3: Example of a Multilayer Perceptron with two hidden layers.

2.2.1 Feedforward Neural Networks

A Multilayer Perceptron is a feedforward neural network composed of an input layer, an output layer, and $N \geq 1$ hidden layers. Its capacity is primarily controlled by the network *depth* (the number of hidden layers) and *width* (the number of neurons per layer). In general, the width may vary across layers; thus, we denote by m_l the number of neurons in hidden layer l.

The term *feedforward* indicates that the computational graph is acyclic: connections transmit information from earlier to later layers, without recurrent connections or lateral connections within the same layer. Given an input x, the network output is computed by *forward propagation*, in which the input is transformed sequentially across layers via affine maps and non-linear activation functions (see Section 2.2.2).

The **MLP** shown in Figure 2.3 takes as input samples in $\mathbb{R}^6$ and is composed of two hidden layers with 4 and 3 neurons, respectively. Its trainable parameters consist of the weight matrices and bias vectors defining the affine transformations at each layer:

- $\mathbf{x} \in \mathbb{R}^6$: input sample;

- $\phi_\ell(\cdot)$: activation function at layer ℓ (possibly varying across layers);
- $\mathbf{W}_1 \in \mathbb{R}^{4\times 6}$, $\mathbf{b}_1 \in \mathbb{R}^4$: weight matrix and bias vector of the first hidden layer;
- $\mathbf{W}_2 \in \mathbb{R}^{3\times 4}$, $\mathbf{b}_2 \in \mathbb{R}^3$: parameters of the second hidden layer;
- $\mathbf{W}_3 \in \mathbb{R}^{1\times 3}$, $b_3 \in \mathbb{R}^1$: parameters of the output layer.

The total number of learnable parameters is therefore

$$(4\cdot 6+4)+(3\cdot 4+3)+(1\cdot 3+1)=47.$$

The forward pass can be written compactly as

$$\mathbf{h}^{(1)} = \phi_1(\mathbf{W}_1\mathbf{x}+\mathbf{b}_1), \tag{2.6}$$

$$\mathbf{h}^{(2)} = \phi_2(\mathbf{W}_2\mathbf{h}^{(1)}+\mathbf{b}_2), \tag{2.7}$$

$$\hat{y} = \psi(\mathbf{W}_3\mathbf{h}^{(2)}+b_3), \tag{2.8}$$

where $\psi(\cdot)$ denotes the *output activation* (e.g., the identity map for regression, or a sigmoid/softmax for classification, depending on the task formulation).

Each hidden layer produces an intermediate representation that can be interpreted as a *learned feature space* for subsequent layers. These representations are not specified manually; instead, they are optimized by minimizing a loss function. Parameter updates are obtained via gradient-based optimization using *backpropagation* (see Section 2.2.3), which propagates error signals from the output layer to earlier layers through the chain rule of calculus.

The term Deep Learning refers to architectures with multiple hidden layers, enabling the composition of many non-linear transformations. Increasing *depth* and *width* enlarges the hypothesis class (i.e., the set of functions representable by the network), which can improve approximation and predictive performance when supported by sufficient data and appropriate regularization. Conversely, overly expressive models trained without adequate regularization may overfit, capturing noise and dataset-specific artifacts rather than task-relevant structure.

A foundational theoretical result underpinning **MLPs** is the *universal approximation theorem*. Informally, for suitable activation functions, a neural network with a single hidden layer and sufficiently many neurons can approximate any continuous function on a compact domain to arbitrary accuracy. More precisely, for a compact set $K \subset \mathbb{R}^d$ and any continuous target function $f^* : K \to \mathbb{R}$, for any $\epsilon > 0$ there exists a one-hidden-layer network f such that

$$\sup_{x \in K} |f(x) - f^*(x)| < \epsilon. \tag{2.9}$$

2.2.2 Activation Functions

Activation functions are *non-linear transformations* applied to the output of each neuron, enabling **MLPs** to model complex, non-linear relationships. Three commonly used functions in literature are:

- **Sigmoid**. A continuous, differentiable function that maps real-valued inputs to $(0, 1)$, often used in the output layer for binary classification task (see Figure 2.4). Its main drawbacks are saturation for large $|x|$ (leading to very small gradients)

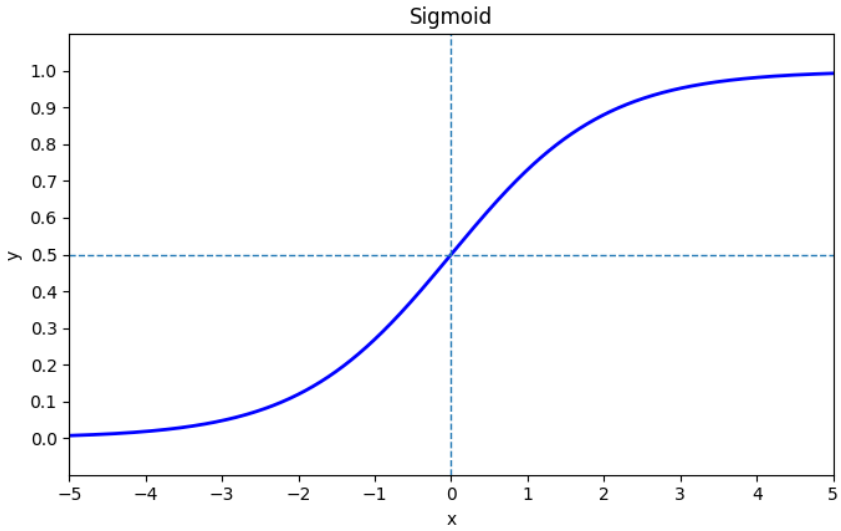

Figure 2.4: Sigmoid activation function.

and non-zero-centered outputs.

$$\sigma(z) = \frac{1}{1+e^{-z}}. \tag{2.10}$$

- **Hyperbolic Tangent (TanH)**. Closely related to the sigmoid, it maps inputs to $(-1,1)$ and is zero-centered, which can improve optimization compared to the former. However, it still saturates for large $|x|$ and can suffer from vanishing gradients (see Figure 2.5).

$$\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}. \tag{2.11}$$

- **Rectified Linear Unit (ReLU)**. It is defined as the positive part of its input, it is

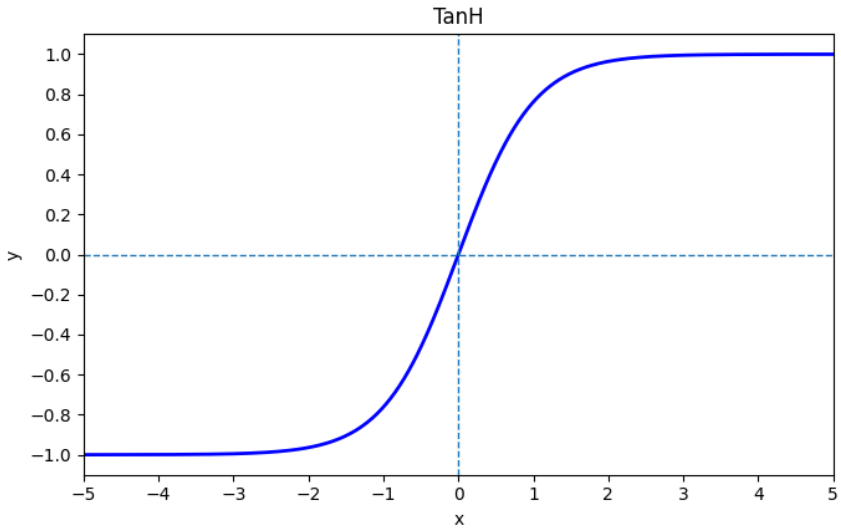

Figure 2.5: Hyperbolic Tangent activation function.

computationally efficient and often accelerates training (see Figure 2.6).

$$\text{ReLU}(z) = \max\{0, z\}. \tag{2.12}$$

Although it is not differentiable at $x = 0$ (handled via sub-gradients in practice), a more relevant limitation is the zero gradient for $x < 0$, which can lead to dead neurons; this motivates variants such as **leaky ReLU**, defined as $g(z) = max\{\alpha z, z\}$, where α is a small value.

Without non-linear activation functions, an **MLP** collapses to a single linear transformation. Ignoring biases for simplicity, a two-layer network becomes

$$f(\mathbf{x}) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} = \mathbf{W}\mathbf{x}, \tag{2.13}$$

where $\mathbf{W} = \mathbf{W}_2\mathbf{W}_1$, since the composition of linear transformations is itself linear.

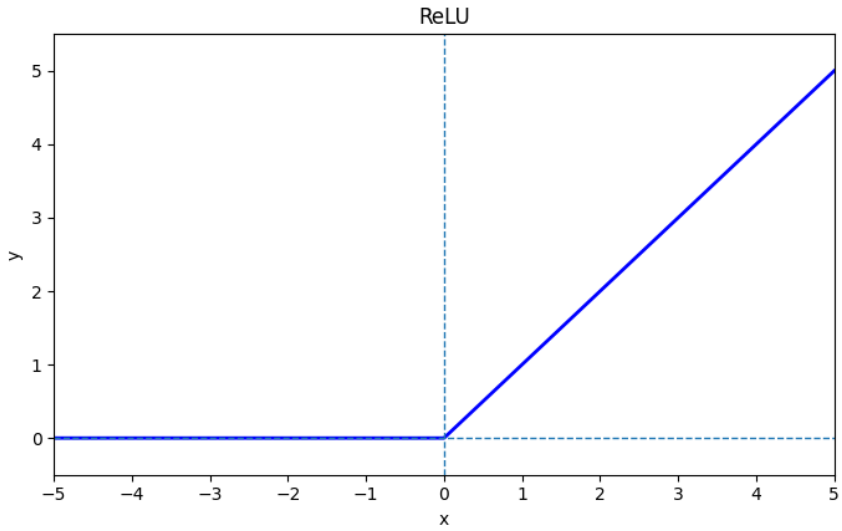

Figure 2.6: ReLU activation function.

2.2.3 Backpropagation

During training, a Deep Learning model learns the set of parameters θ that minimizes a loss function over the training data. The optimization step itself is typically performed by gradient-based methods, while backpropagation provides the gradients required for that update. More precisely, backpropagation computes the derivative of the loss function with respect to each trainable parameter in the network by repeatedly applying the chain rule from the output layer toward the input layer [9].

The relevance of this procedure follows from the compositional structure of neural networks. A network can be viewed as a sequence of simple transformations, where the output of one layer becomes the input of the next. Since the loss depends on the final output, and the final output depends on all preceding layers, the derivative of the loss with respect to an internal parameter can be obtained by decomposing the global

dependency into local derivatives. This is exactly the role of the chain rule.

Consider a neuron or layer i with output o_i, parameters W_i and b_i, and input o_{i-1}. In order to propagate the error backward through this component, it is sufficient to compute the local derivatives of its output with respect to its parameters and its input, namely

$$\frac{\partial o_i}{\partial W_i}, \qquad \frac{\partial o_i}{\partial b_i}, \qquad \frac{\partial o_i}{\partial o_{i-1}}.$$

Once the gradient of the loss with respect to the output o_i is available, these local terms make it possible to obtain the gradients needed for both parameter updates and further propagation:

$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial o_i}\frac{\partial o_i}{\partial W_i}, \qquad \frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial o_i}\frac{\partial o_i}{\partial b_i}, \qquad \frac{\partial L}{\partial o_{i-1}} = \frac{\partial L}{\partial o_i}\frac{\partial o_i}{\partial o_{i-1}}.$$

Therefore, backpropagation starts from the output layer, where the loss is explicitly defined, and recursively transmits the gradient to all preceding layers. At each step, the gradient with respect to the current layer output is combined with the corresponding local derivatives, producing the gradients of the parameters in that layer and the gradient to be passed to the previous one. In this way, the error signal is propagated through the entire network in reverse order. After all gradients have been computed, the parameters are updated through gradient descent or one of its variants. Backpropagation is therefore not an optimization algorithm by itself, but the mechanism that makes gradient-based learning feasible in deep neural networks.

2.2.4 Convolutional Neural Networks

Images are commonly represented as 3D tensors $T \in \mathbb{R}^{H \times W \times C}$, where H and W denote the height and width of the image, and C represents the number of channels (also called depth; i.e. an RGB image has 3 channels).

Using traditional **MLPs** for processing images presents several challenges:

1. **Fixed input dimensions**. The network requires inputs of a fixed size, yet images can vary widely in their dimensions, requiring additional pre-processing steps;

2. **Loss of spatial structure**. The 3D image tensor must be reshaped into a 1D tensor, often via a raster scan. The flattening process disrupts the inherent spatial organization of pixels, losing critical information about local structures;

3. **Disruption of local pixel correlations**. Images exhibit strong local correlations, flattening the image into a vector obscures these spatial relationships, which are essential for identifying patterns and meaningful features (e.g. color, texture, edges);

4. **Explosive growth in parameters**. In **MLPs** every input neuron is connected to every other one in the next layer, causing the number of parameters to grow rapidly with the input size. Given the size of an image in pixels, often on multiple channels, this leads to an excessive number of parameters, resulting in high computational costs.

When using **MLPs** for processing images, each neuron in a layer attempts to model the correlation between a given pixel and all other pixels in the image. In reality,

useful features in images primarily emerge from local regions (e.g. edges, textures, shapes). This insight underscores the need for sparse connectivity, where each neuron is connected only to a limited subset of them in the previous layer focusing on local rather than global patterns.

Convolutional Neural Networks (**CNNs**) address these challenges by incorporating two key architectural innovations:

- **Local receptive fields**. Instead of processing the entire image at once, **CNNs** use *small localized filters* that scan across the image. This design preserves the spatial structure and local correlations between pixels;
- **Weight sharing**. The same set of weights of a filter is applied across different regions of the image, which significantly reduces the number of parameters compared to **MLPs**.

CNNs remain fully differentiable, with learnable weights, biases, and non-linear activation functions, making them trainable via gradient-based optimization. **CNNs** use the convolution operator instead of a purely linear transformation, making them more suited for processing image data.

The *convolution operator* is the fundamental operation for feature extraction in **CNNs**. It combines an input image I and a kernel K through a series of multiplications and summations to generate an output feature map. The standard definition of discrete convolution is given by (in the equation the image I has only 1 channel; the kernel K

is flipped both horizontally and vertically before it is applied to the input image):

$$F(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n) \cdot K(m,n). \tag{2.14}$$

In most Deep Learning libraries, the operation described in Equation 2.14 is implemented as *cross-correlation*, which omits the kernel flipping, defined as (this difference is largely inconsequential in practice, as the kernels are learned during training; the network adjusts the weights to capture the desired features regardless of whether the kernel is flipped or not):

$$F(i,j) = \sum_m \sum_n I(i+m, j+n) \cdot K(m,n). \tag{2.15}$$

Figure 2.7 shows an example of convolution between an image and a kernel, both operating on a single channel. The kernel is referenced from its top-left corner. This convention ensures that when the kernel is applied, it extracts a patch from the image corresponding to its size. The patch can then be unrolled into a vector, and a dot product is computed with the kernel, maintaining efficiency through positive coordinate indices.

The result of the convolution operation tells how *similar* the local region of the image is to the learned pattern encoded in the kernel. A high output value suggests that the image patch closely matches the pattern (e.g. edge, color gradient, shape), while a low value indicates little similarity. Convolutional layers form the core building blocks of **CNNs**, and their structure is designed to effectively capture local spatial patterns from images.

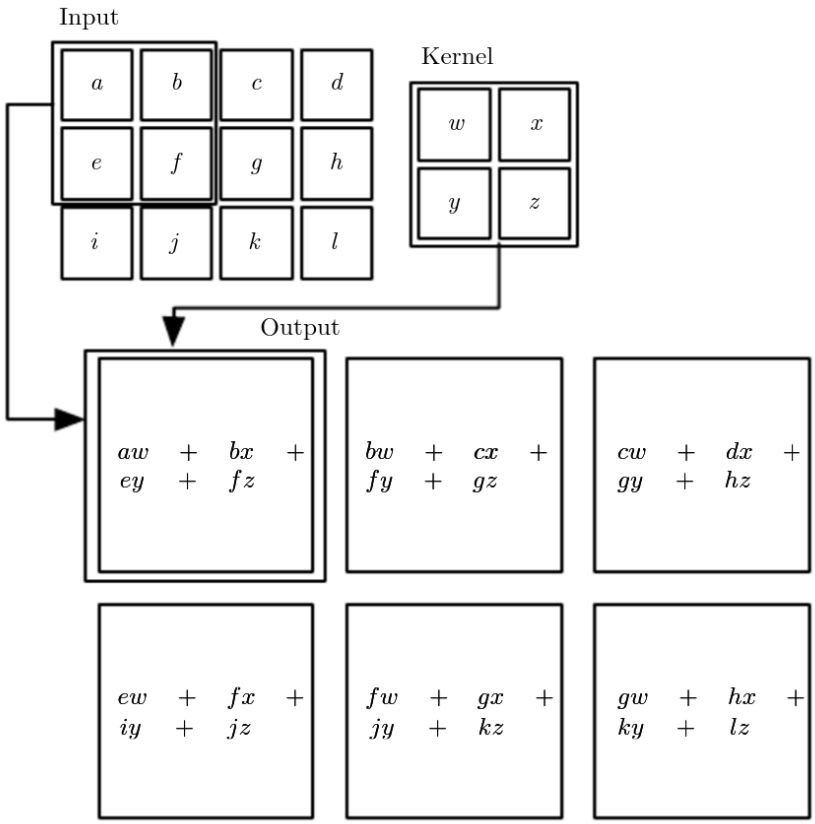

Figure 2.7: Example of 2D convolution without kernel-flipping on an image with a single channel. Figure by courtesy of [3].

2.2.5 ResNet Architecture

A natural question in Deep Learning is whether better representations can be obtained simply by increasing network depth. In principle, deeper models should be more expressive; in practice, however, optimization becomes increasingly difficult as layers are added. Early deep architectures were strongly affected by *vanishing and exploding gradients*, which made training unstable. Although careful weight initialization and normalization layers, such as Batch Normalization (**BN**), have greatly mitigated this

issue, increasing depth still introduces another important difficulty: the *degradation problem* [4].

The degradation problem refers to the empirical observation that, beyond a certain depth, adding more layers may worsen both training and test performance. This behavior cannot be explained solely by *overfitting*, since the deeper model often exhibits higher training error than its shallower counterpart. Therefore, the issue is not only one of model capacity, but also of optimization.

Residual Networks (**ResNets**) address this problem through the idea of *residual learning*. Instead of forcing a stack of layers to directly approximate a desired mapping $\mathrm{H}(x)$, the network is trained to approximate a residual function with respect to the input:

$$\mathrm{F}(x) := \mathrm{H}(x) - x. \tag{2.16}$$

The original mapping can then be rewritten as

$$\mathrm{H}(x) = \mathrm{F}(x) + x. \tag{2.17}$$

This reformulation makes the learning problem easier in many cases. In particular, if the optimal transformation is close to the identity mapping, the residual branch only needs to learn a small correction, rather than reconstructing the identity through multiple non-linear layers.

The basic building block of a ResNet is therefore a *residual block*, in which the input is combined with the output of a sequence of learnable layers through a shortcut connection. In its simplest form, the block can be written as

$$y = \mathrm{F}(x, \{W_i\}) + x, \tag{2.18}$$

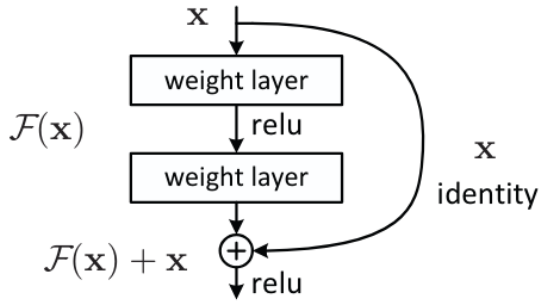

Figure 2.8: A ResNet building block. Figure by courtesy of [4].

where x is the input to the block, y is the output, and $\mathrm{F}(x, \{W_i\})$ denotes the residual mapping parameterized by the trainable weights $\{W_i\}$. The shortcut path performs an element-wise addition and, when the input and output have the same dimensions, introduces neither additional parameters nor significant computational overhead.

For the residual addition to be valid, the dimensions of x and $\mathrm{F}(x)$ must match. When this condition does not hold, for example because the number of channels changes or spatial downsampling is performed, the shortcut can be replaced by a linear projection:

$$y = \mathrm{F}(x, \{W_i\}) + W_s x, \tag{2.19}$$

where W_s is typically implemented as a 1×1 convolution, possibly with stride 2. This variant is commonly referred to as a *projection shortcut*. In all other cases, identity shortcuts are generally preferred, since they preserve the simplicity and efficiency of the residual formulation.

Although the residual framework can be expressed in terms of generic non-linear layers, it is particularly effective in **CNNs**. In this setting, $\mathrm{F}(\cdot)$ usually consists of a small stack of convolutional layers, each followed by Batch Normalization and a non-linear

activation function such as ReLU (see Section 2.2.2). The shortcut connection then adds the input feature map to the transformed feature map channel-wise.

The original ResNet architecture follows a simple design principle. Convolutional layers mostly use 3×3 filters, and layers operating at the same spatial resolution typically use the same number of channels. When the feature-map resolution is reduced, usually by a factor of two, the number of channels is increased in order to maintain an adequate representational capacity. Downsampling is performed directly by strided convolutions, while the classifier at the end of the network is usually composed of a global average pooling layer followed by a fully connected layer.

The main contribution of ResNets is therefore not a radically different convolutional operator, but an architectural reformulation that makes very deep networks easier to optimize. By facilitating gradient propagation and simplifying the learning of identity mappings, residual connections allow the construction of substantially deeper models with improved training behavior and stronger empirical performance [4].

2.3 Continual Learning

In standard supervised learning we usually make (often implicitly) the *i.i.d.* assumption: training samples are drawn from a single, fixed distribution, and we can revisit the same dataset for multiple epochs. Under this setting, gradient-based optimization searches for one set of parameters that minimizes the loss over the whole training distribution. Continual Learning (**CL**), also known as *Lifelong Learning*, relaxes this

setting. Data arrive sequentially and the underlying distribution is allowed to change over time (new tasks, new domains, new classes, or a mixture of these). In practice, only a portion of the data stream is available at any moment: when the model moves on to a new task, past data may be unavailable due to storage constraints, privacy limitations, or because retraining from scratch would be too expensive in a deployed system.

We model **CL** as a sequence of T tasks with different distributions:

$$D = \{D_1, \ldots, D_T\}, \qquad D_t = \{(x_{i,t}, y_{i,t})\}_{i=1}^{n_t},$$

where $x_{i,t} \in X$ is the input and $y_{i,t} \in Y$ is the corresponding label.

In supervised classification, the goal is to learn a function $f_\theta : X \times T \to Y$ — parameterized by learnable weights θ — that maps an input-task pair (x, t) to the correct label y for samples $(x, y) \sim D_t$. At a given point in the stream (current task t_c), a common formulation is to minimize the *cumulative risk* over the tasks observed so far:

$$\theta^* = \arg\min_\theta \sum_{t=1}^{t_c} \mathbb{E}_{(x,y)\sim D_t} \big[\ell\big(y, f_\theta(x, t)\big)\big], \tag{2.20}$$

where $\ell(\cdot)$ is the loss function.

Compared to the classical *i.i.d.* setting, this formulation immediately introduces a few non-trivial issues:

- **Non-i.i.d. stream**. Each task corresponds to a different distribution, so the gradients that are optimal for the current task can conflict with those that were optimal for past tasks.

- **Forgetting vs. transfer**. Ideally, knowledge should accumulate: previous experience should help future learning (positive transfer) without degrading past performance.

- **Boundaries may be unknown**. In many realistic settings, the model is not told when the task changes (task-free/online continual learning), and shifts can be gradual rather than abrupt.

2.3.1 Catastrophic Forgetting

The main failure mode in Continual Learning is *catastrophic forgetting*: after learning a new task, the model's performance on previously learned tasks can drop sharply [10]. The root cause is not simply limited capacity, but the *sequential* training procedure itself. When we optimize on the current task only, parameter updates move θ toward the current loss minimum, with no incentive to remain in regions that also worked well for past tasks.

A useful intuition comes from distribution shift. If the data distribution changes, the *loss landscape changes* with it: minima move, and gradient directions change over time. In a neural network, each task tends to rely on a specific configuration of internal features (and therefore parameters). When a new task reuses the same parameters, updating them to fit the new distribution can overwrite feature representations that were important for earlier tasks.

Continual Learning is essentially a balancing act between two competing needs:

- **Plasticity**: the ability to adapt quickly to new tasks.

- **Stability**: the ability to preserve previously acquired knowledge.

This tension is often referred to as the *stability–plasticity dilemma* [11]. Too much plasticity leads to fast adaptation but severe forgetting; too much stability preserves old tasks but makes the model rigid and unable to learn.

If we could store all historical data and train on the union of tasks (interleaving samples from all tasks), forgetting would be largely mitigated. However, that solution defeats the continual setting: storing past data may be infeasible, and retraining from scratch or maintaining a growing training set may be too costly in time, compute, or energy.

Most Continual Learning methods can be grouped into three families, each one with unique traits:

- **Rehearsal-based methods**: replay a small buffer of real samples (or synthetic samples) from past tasks while learning the current one.
- **Regularization-based methods**: penalize changes to parameters deemed important for previous tasks.
- **Architectural methods**: allocate separate capacity (modules/heads/masks) to reduce interference between tasks.

2.3.2 Scenarios

To make evaluation comparable across papers, it is common to distinguish three Continual Learning scenarios based on what is available at test time [1]. The key

variable is whether the model is given the *task identity* (task-ID) at inference time. Table 2.1 summarizes the three scenarios.

1. **Task-Incremental Learning (Task-IL)**. The task-ID is provided at test time. This makes the setting easier because the model can route the input to task-specific components (e.g., a multi-headed classifier where each head corresponds to a task). The learner is not required to infer which task the sample belongs to.

2. **Domain-Incremental Learning (Domain-IL)**. The task-ID is not provided, but tasks typically share the same label space and differ mainly in input distribution (domain shift). Evaluation is usually performed on the current domain/task. The model must be robust to distribution changes without explicit task cues.

3. **Class-Incremental Learning (Class-IL)**. This is the most challenging scenario. Each task introduces new classes (often disjoint label sets), and at test time the model must classify samples among *all* classes seen so far, without knowing the task-ID. In practice, this forces a *single-headed* classifier shared across tasks, which amplifies interference and makes forgetting harder to control.

Scenario	***Required at test time***
Task-IL	Solve tasks so far, task-ID provided
Domain-IL	Solve tasks so far, task-ID not provided
Class-IL	Solve tasks so far and infer task-ID

Table 2.1: Three continual learning scenarios and their features. Table by courtesy of [1].

In the remainder of this work, we focus primarily on *Class-Incremental Learning*. This scenario is the closest to the practical requirement of expanding a classifier over time (new categories appear, old ones must remain recognized) while operating without a test-time oracle, and it exposes the *stability–plasticity trade-off* in its most explicit form.

2.4 Spiking Neural Networks

Spiking Neural Networks (**SNNs**) are a family of neural models where information is represented and propagated through *spikes* (discrete events in time) rather than continuous activations. From a biological perspective, this is closer to how real neurons communicate; from an engineering perspective, it enables *event-driven* computation, since neurons only emit a signal when their internal state crosses a threshold [5, 12].

An **SNN** processes inputs over time. Instead of producing a single forward pass output, the network evolves its internal states (membrane potentials) across a temporal horizon and emits *spike trains*. A spike train can be represented either as a set of firing times $\{t_k\}$, or equivalently as

$$s(t) = \sum_k \delta(t - t_k), \tag{2.21}$$

where $\delta(\cdot)$ is the Dirac delta function. In practice, especially for training, time is discretized and each neuron emits a binary signal $s[t] \in \{0, 1\}$ at each timestep t.

SNNs can be interpreted as a particular form of Recurrent Neural Network (**RNN**): the state of each neuron is a dynamical variable that integrates past inputs, and spikes introduce non-linear, discontinuous transitions. This temporal structure makes **SNNs**

suitable for sequential data and for neuromorphic hardware implementations. This recurrent interpretation is also useful from an optimization perspective: once unfolded in time, **SNNs** can be treated with many of the same tools used for recurrent models, although the binary nature of spikes makes the learning problem substantially harder [13].

2.4.1 Leaky Integrate-and-Fire (LIF) Neuron Model

The most common neuron model used in modern **SNNs** is the *Leaky Integrate-and-Fire* (**LIF**) neuron, due to its close connection to a Resistor–Capacitor (**RC**) circuit [5], and its simplicity as it lays in between a common *Deep Learning* neuron and the "*gold standard*" for bio-physically realistic neuron modeling (Hodgkin-Huxley neuron) [14]. Figure 2.9 shows the connection between **LIF** neuron and **RC** circuit. Each neuron maintains a *membrane potential* $u(t)$ that integrates the input current and decays over time (leak). When $u(t)$ crosses a threshold, the neuron emits a spike and the membrane potential is reset.

A standard **LIF** dynamics can be written as:

$$\tau_m \frac{d\,u(t)}{d\,t} = -\big(u(t) - u_{\text{rest}}\big) + R\,I(t), \tag{2.22}$$

with the spike emission rule:

$$s(t) = \mathbb{I}\big(u(t) \geq \vartheta\big), \tag{2.23}$$

and a reset mechanism (one of the common choices):

$$\text{if } s(t) = 1 \text{ then } u(t) \leftarrow u_{\text{reset}}. \tag{2.24}$$

Where:

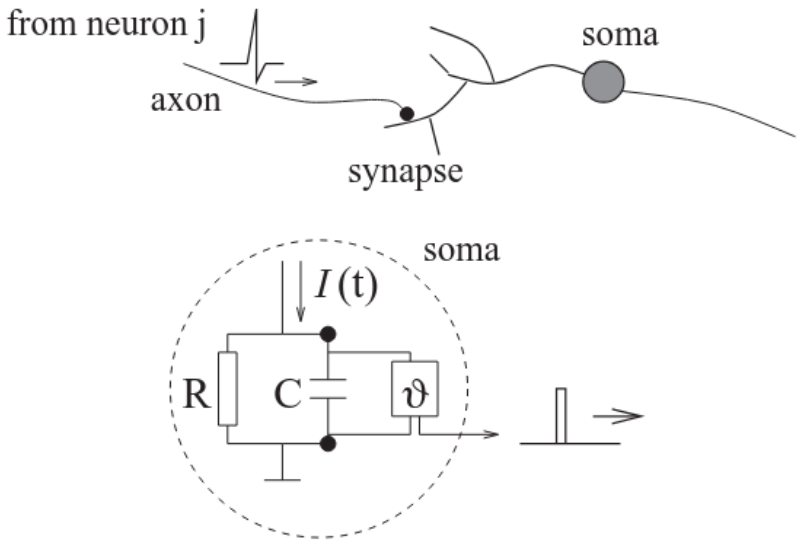

Figure 2.9: Connection between LIF neuron and RC circuit. The soma in the neuron works as a RC circuit. Figure adapted from [5].

- τ_m is the membrane time constant.
- u_{rest} is the membrane resting potential.
- R is a resistance term (usually absorbed in the definition of the input current).
- $I(t)$ is the input current (from synapses and/or external stimulus).
- ϑ is the firing threshold.
- u_{reset} is the membrane reset potential.

For simulations and gradient-based training, (2.22) is typically discretized with a timestep Δt:

$$u[t+1] = \alpha\, u[t] + (1-\alpha)\, u_{\text{rest}} + R\,\Delta t\, I[t] - s[t](u[t] - u_{\text{reset}}), \tag{2.25}$$

where $\alpha = \exp(-\Delta t / \tau_m)$ and the spike is

$$s[t] = \mathbb{I}\big(u[t] \geq \vartheta\big). \tag{2.26}$$

The last term in (2.25) implements the *reset* (only active when a spike occurs). Other equivalent implementations are possible (i.e.: hard reset $u[t+1] \leftarrow u_{\text{reset}}$ when $s[t] = 1$, or subtracting the threshold after emission). In this work we adopt the *subtract* reset mechanism.

2.4.2 Non-differentiability of Spikes

Training **SNNs** with gradient descent is not straightforward because the network must solve both a *temporal credit assignment* problem and a *non-differentiability* problem. On the one hand, the internal state evolves over time, so optimization typically requires unfolding the dynamics and propagating gradients across multiple time-steps, as in recurrent models. On the other hand, spikes are generated through a hard threshold, which makes the activation function discontinuous and prevents the direct application of standard backpropagation [13]. The spike generation in (2.26) is a hard threshold:

$$s[t] = \mathbb{I}(u[t] - \vartheta \geq 0),$$

which is equivalent to a *Heaviside step function.* Its derivative is zero almost everywhere and undefined at the threshold:

$$\frac{\partial s}{\partial u} = \begin{cases} 0 & \text{if } u \neq \vartheta, \\ \text{undefined} & \text{if } u = \vartheta. \end{cases}$$

In distributional terms, the derivative can be associated with a *Dirac delta* function centered at ϑ (unit impulse), which is not usable in standard backpropagation.

Assume we want to optimize a loss $\mathscr{L}$ that depends on spikes. Gradient descent requires:

$$\frac{\partial \mathscr{L}}{\partial u[t]} = \frac{\partial \mathscr{L}}{\partial s[t]} \cdot \frac{\partial s[t]}{\partial u[t]}. \tag{2.27}$$

Since

$$\frac{\partial s[t]}{\partial u[t]}$$

is either 0 or undefined, gradients either vanish (no learning signal) or become numerically unstable around threshold crossings. This issue compounds over time because **SNNs** are trained over sequences: the membrane state creates temporal dependencies, so training typically requires Backpropagation Through Time (**BPTT**), which already suffers from *vanishing/exploding gradients* even in smooth **RNNs**.

A standard workaround is to keep the *forward* computation unchanged (hard spikes), but replace the problematic derivative in the *backward* pass with a smooth proxy. Concretely, one defines:

$$\frac{\partial s[t]}{\partial u[t]} \approx \frac{\partial \tilde{s}[t]}{\partial u[t]}, \tag{2.28}$$

where $\tilde{s}$ is a differentiable function that approximates the step around the threshold ϑ.

Standard **SNNs** surrogate literature include the following:

- **Arc-tangent (aTan).** A smooth, bell-shaped surrogate derivative centered at the threshold:

$$\frac{\partial \tilde{s}[t]}{\partial u[t]} = \frac{\alpha}{2} \frac{1}{1 + \left(\frac{\pi\alpha}{2}(u[t] - \vartheta)\right)^2}, \tag{2.29}$$

where $\alpha > 0$ controls the slope (larger α yields a sharper peak).

- **Sigmoid.** Using $\sigma(x) = \frac{1}{1+\exp(-x)}$ as a smooth approximation of the step:

$$\frac{\partial \tilde{s}[t]}{\partial u[t]} = k\,\sigma(k(u[t]-\vartheta))(1-\sigma(k(u[t]-\vartheta))), \tag{2.30}$$

  where $k > 0$ controls the steepness around ϑ.

- **Piecewise-linear (triangular).** A compact-support surrogate derivative (non-zero only in a neighborhood of the threshold):

$$\frac{\partial \tilde{s}[t]}{\partial u[t]} = \max\bigl(0,\ 1-\beta\,|u[t]-\vartheta|\bigr), \tag{2.31}$$

  where $\beta > 0$ controls the width of the non-zero region (larger β implies a narrower support).

- **Straight-through estimator (STE).** A simple approximation that passes gradients through unchanged (very aggressive option, rarely used):

$$\frac{\partial \tilde{s}[t]}{\partial u[t]} = 1. \tag{2.32}$$

Figure 2.10 shows a comparison plot of the previously listed surrogate gradient functions. With surrogate gradients, the neuron remains event-driven in the forward pass (binary spikes), while the backward pass becomes differentiable enough to optimize synaptic weights using standard optimizers. Most modern supervised **SNN** training pipelines therefore look like this:

1. Simulate **LIF** dynamics forward in time, producing spikes $s[t]$.

2. Compute a task loss $\mathscr{L}$.

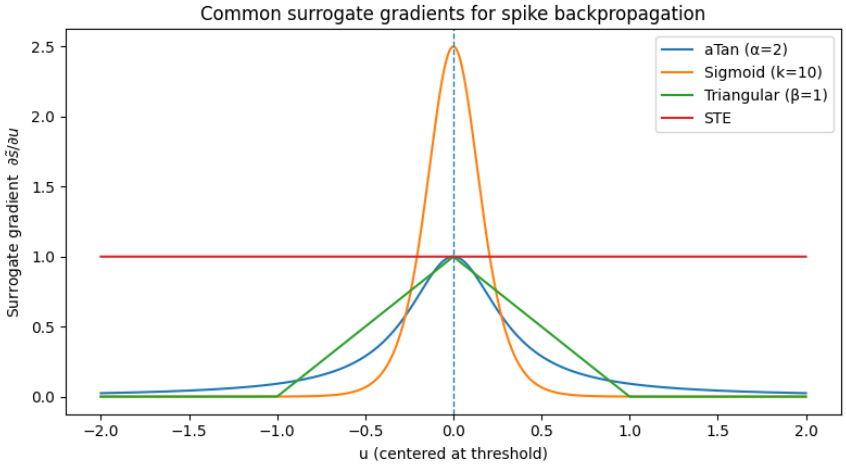

Figure 2.10: Comparison between common surrogate gradient functions used in LIF neuron.

3. Apply **BPTT**, using a surrogate gradient for $\partial s / \partial u$.

This approach is currently the most common way to directly train deep **SNNs**, and it is the default baseline when comparing spiking models to standard **ANNs** on supervised benchmarks.

Chapter 3

Related Works

This chapter reviews the literature that is most relevant to our work, both from the perspective of *Continual Learning* and with respect to the design principles behind the proposed method, **STAER**. We begin by discussing the main families of **CL** strategies, with a particular focus on rehearsal-based methods under bounded-memory constraints. We then introduce the *Soft-DTW divergence*, which provides the temporal alignment component used in our approach.

3.1 Continual Approaches

A large portion of Deep Learning methods is developed under a *multi-task* (or joint-training) setting, where data from all tasks are assumed to be available at the same time and optimization is carried out over the union of all corresponding datasets. Under this assumption, joint training usually provides strong performance, since the model can continuously revisit all tasks during optimization. In practice, however, this setting

is often unrealistic: it requires storing all past data and assumes that the full set of tasks is known in advance. These assumptions break down in many real scenarios, where tasks arrive *sequentially* and their order is *not controlled* by anyone.

Continual Learning addresses this more realistic regime by training a model on a non-stationary sequence of data/tasks while trying to preserve knowledge acquired in earlier stages. The central difficulty is *catastrophic forgetting*, namely the loss of performance on previously learned tasks caused by adaptation to new data. Over time, several families of methods have been proposed to reduce forgetting, including *regularization-based* approaches, *architectural* (or *parameter-isolation*) strategies, and *rehearsal-based* methods. In this chapter, we concentrate on the approaches that are most closely related to the method proposed in this thesis.

For our setting, a particularly relevant family is represented by *rehearsal-based* methods, and especially by their *replay-based* variants. These approaches store a bounded set of past samples (or exemplars) and reuse them together with current data during training. In this way, they approximate joint training over past and present distributions while respecting constant-memory constraints. This makes them especially suitable for *General Continual Learning* (**GCL**), where task boundaries may be unavailable and buffer management is preferably task-agnostic. Replay can also be combined with *distillation* or consistency constraints, yielding hybrid objectives that preserve not only target labels but also richer signals such as intermediate representations or logits.

Since **STAER** is designed for sequential learning under bounded memory and does not rely on strong assumptions about task structure, rehearsal-based methods provide

the most natural reference point, both as baselines and as building blocks. For this reason, the remainder of this section introduces the *Class-Incremental* exemplar-based method **iCaRL**, to then focus on *Experience Replay* (**ER**) and on its dark-knowledge variants, **DER** and **DER++**, which are strong and widely used foundations in modern Continual Learning.

3.1.1 Incremental Classifier and Representation Learning

incremental **C**lassifier and **R**epresentation **L**earning (**iCaRL**) is a foundational rehearsal-based method for *Class-Incremental* Learning, where classes are introduced sequentially and the model must retain the ability to classify among all classes observed so far under a bounded memory budget [15]. Unlike standard replay schemes that rely directly on the classifier head at inference time, **iCaRL** combines exemplar rehearsal, distillation, and a prototype-based prediction rule. In this way, it jointly updates the feature representation and preserves a compact summary of previously seen classes.

Let $\phi_\theta(x)$ denote the feature extractor learned by the network, and let P^y be the exemplar set associated with class y. If t classes have been observed and the total memory budget is fixed to K, **iCaRL** allocates approximately $m = K/t$ exemplars per class, reducing older exemplar sets whenever new classes are introduced [15]. Classification is then performed through a *nearest-mean-of-exemplars* rule. For each class y, a prototype is computed as

$$\mu_y = \frac{1}{|P^y|} \sum_{p \in P^y} \phi_\theta(p), \tag{3.1}$$

and a test sample x is assigned to the closest prototype:

$$y^* = \arg\min_{y \in \{1,\dots,t\}} \|\phi_\theta(x) - \mu_y\|_2. \tag{3.2}$$

This choice is important in the incremental setting because the representation changes after each update. Using class prototypes derived from stored exemplars makes the prediction rule less dependent on a fixed set of output weights, whose semantics may drift as new classes are incorporated. In this sense, the classifier remains tied to the current representation, rather than to parameters learned at an earlier stage.

When a new batch of classes becomes available, **iCaRL** updates the representation on the union of current training examples and stored exemplars [15]. To mitigate forgetting, the supervised objective on new classes is complemented by a *distillation* term on previous ones, which encourages the updated network to preserve its earlier responses for old classes. In abstract form, the training objective can be written as

$$\mathscr{L} = \mathscr{L}_{\text{sup}} + \lambda \mathscr{L}_{\text{dist}}, \tag{3.3}$$

where $\mathscr{L}_{\text{sup}}$ fits the current classes and $\mathscr{L}_{\text{dist}}$ constrains the network to remain consistent with the pre-update model on past classes.

Exemplar management is another central component of the method. Rather than sampling stored examples uniformly at random, **iCaRL** constructs, for each class, a prioritized list of exemplars chosen so that the average feature vector of the selected samples approximates the full class mean as closely as possible [15]. This procedure allows the memory assigned to older classes to be reduced by simple truncation when new classes arrive, while still preserving a representative approximation of each class prototype. Since exemplars are stored in the input space rather than in feature space, their

representations can be recomputed after each update and therefore remain aligned with the current encoder.

Although **iCaRL** is a crucial precursor to modern replay methods, approaches such as **ER**, **DER**, and **DER++** provide a more direct reference for the task-agnostic **GCL** setting considered in this work.

3.1.2 Experience Replay

In Continual Learning, optimization over a non-stationary data stream generally causes *catastrophic forgetting*, i.e., a progressive degradation of performance on previously observed data as the model is updated on new samples. Early studies in *connectionist memory* already identified this issue [16]. In particular, analysis of sequential training in **MLPs** showed that, in the absence of a mechanism to recover earlier patterns for rehearsal, information learned with high confidence can be quickly overwritten by subsequent updates. A direct way to counter this effect is *rehearsal* (and *pseudo-rehearsal*), where older information—either real or approximated—is interleaved during the learning of new data [17].

Experience **R**eplay (**ER**) can be viewed as the modern Deep Learning realization of rehearsal. The method maintains a *bounded memory buffer* of previously seen examples and mixes them with the current mini-batch during optimization. **ER** is illustrated in Figure 3.1 and detailed in Algorithm 1.

Let the data stream produce samples (x_n, y_n) during each task $n \in \{1, \dots, N\}$. **ER** keeps

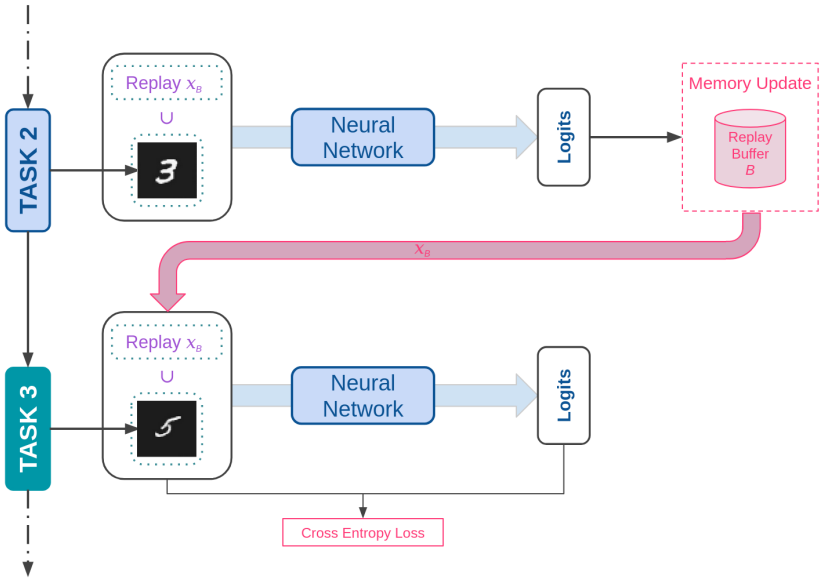

Figure 3.1: Overview of ER method. Samples from previous tasks are stored in a memory buffer and replayed together with current-task data when training on a new task, enabling the shared network to mitigate catastrophic forgetting.

a finite memory $\mathscr{B}$ (replay buffer) with capacity $|\mathscr{B}|$. At training step n:

1. Sample the current mini-batch $B_n \subset \{(x_n, y_n)\}$.

2. Sample a replay mini-batch $B_{\mathscr{B}} \subset \{(x_{\mathscr{B}}, y_{\mathscr{B}})\} \sim \mathscr{B}$.

3. Concatenate the two and update the parameters on the mixed batch $B = B_n \cup B_{\mathscr{B}}$.

From an optimization viewpoint, replay makes the gradient a better proxy for the joint risk over past and current data, thus reducing destructive interference between sequential updates. The same general idea—augmenting the current training signal with information from previous samples—is also used in *exemplar-based* rehearsal

Algorithm 1 ER algorithm

Input: (x_n, y_n) current mini-batch at step n; replay buffer $\mathscr{B}$ with capacity $|\mathscr{B}|$; $(x_{\mathscr{B}}, y_{\mathscr{B}})$ buffer mini-batch.
$\mathscr{B} \leftarrow \emptyset$
for $n \in \{1, \ldots, N\}$ **do**
    Training
    $\mathscr{L} \leftarrow \mathscr{L}_{CE}\Big(f_\theta\big([x_n, x_{\mathscr{B}}]\big), [y_n, y_{\mathscr{B}}]\Big)$ ▷ CE on new+buffer
    Update buffer
    $x_{\mathscr{B}} \leftarrow x_n, \quad y_{\mathscr{B}} \leftarrow y_n$ ▷ Update examples
end for

methods [15]. One optimization step is then:

$$\theta \leftarrow \theta - \eta \nabla_\theta \Big(\frac{1}{|B|} \sum_{(x,y)\in B} \ell\big(y, f_\theta(x)\big) \Big). \tag{3.4}$$

Because the memory budget must remain fixed, **ER** requires a replacement policy when the buffer is full, so as to satisfy the constant-memory constraints of General Continual Learning. A common and effective choice is *Reservoir sampling* [18], which is task-agnostic and streaming-friendly. With reservoir sampling, each incoming example has the same probability of being retained in the buffer, independently of task boundaries.

3.1.3 Dark Experience Replay

Dark **E**xperience **R**eplay (**DER**) is a **CL** method that combines *rehearsal*, *knowledge distillation*, and regularization by matching the network's logits observed along the optimization trajectory (i.e., past experience), thereby encouraging consistency with its previous behavior [19].

A continual classification problem is divided into N tasks. During each task $n \in \{1,\dots,N\}$, input samples x_n and labels y_n are drawn *i.i.d.* from a distribution D_n. A function f, parameterized by θ, is trained sequentially, one task at a time. Let $h_\theta(\cdot)$ denote the output logits, and let $f_\theta(\cdot)$ be the corresponding probability distribution over classes.

The objective is to learn a model that can correctly classify examples from all tasks observed so far, at any point in training, i.e., from tasks in $\{1,\dots,n_c\}$:

$$\arg\min_\theta \sum_{n=1}^{n_c} \mathscr{L}_n = \sum_{n=1}^{n_c} \mathbb{E}_{(x_n,y_n)\sim D_n}\Big[\ell\big(y_n, f_\theta(x_n)\big)\Big]. \tag{3.5}$$

This is difficult because, once task n_c is being learned, data from previous tasks are no longer available. Therefore, the model must search for a good configuration of θ with respect to $\mathscr{L}_{1,\dots,n_c}$ without direct access to D_n for $n \in \{1,\dots,n_c-1\}$. Ideally, the network should fit the current task while preserving its past behavior on old data (i.e., reproducing earlier responses on previously seen samples). To preserve knowledge from earlier tasks, one can consider the following objective:

$$\mathscr{L}_{n_c} + \alpha \sum_{n=1}^{n_c-1} \mathbb{E}_{(x_n,y_n)\sim D_n}\Big[D_{KL}\big(f_{\theta_n^*}(x_n)\|f_\theta(x_n)\big)\Big], \tag{3.6}$$

where θ_n^* denotes the parameters obtained at the end of task n, and α balances the two terms.

The distillation term above would require access to previous-task datasets D_n, which are assumed unavailable. To address this limitation, **DER** introduces a replay buffer $\mathscr{B}_n$ for task n that stores past experiences using the network logits $z = h_{\theta_n}(\cdot)$ (dark knowledge) instead of the labels:

$$\mathscr{L}_{n_c} + \alpha \sum_{n=1}^{n_c-1} \mathbb{E}_{(x_{\mathscr{B}_n},z)\sim \mathscr{B}_n}\Big[D_{KL}\big(\mathrm{softmax}(z)\|f_\theta(x_{\mathscr{B}_n})\big)\Big]. \tag{3.7}$$

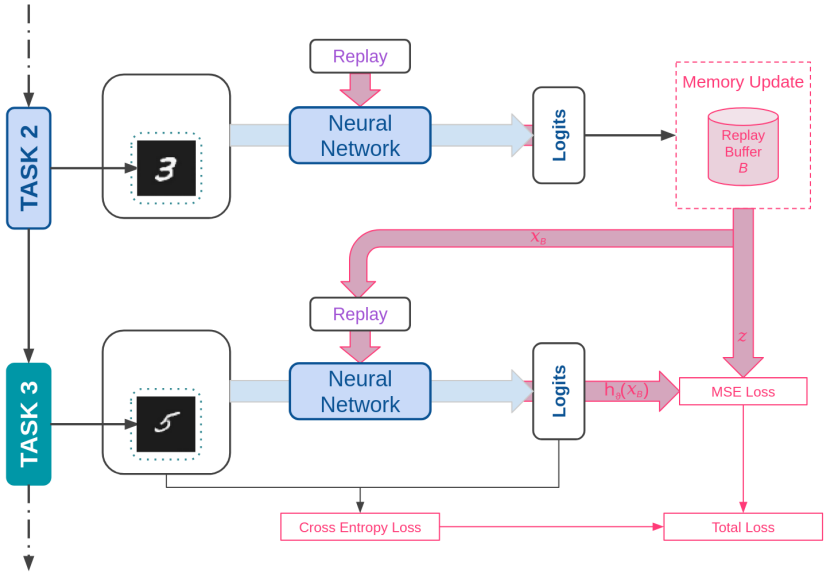

Figure 3.2: Overview of DER method. Samples from previous tasks are stored in a memory buffer together with their corresponding logits and replayed during training on a new task. In addition to the cross-entropy loss on the current-task data, an MSE loss is applied between the stored logits and the current model outputs on replayed samples, enabling the shared network to preserve prior knowledge and mitigate catastrophic forgetting.

By adopting *Reservoir sampling* [18], the buffer can be filled in a task-agnostic manner as training proceeds, without relying on task boundaries and while remaining compatible with the general continual learning setting. If $|\mathscr{B}|$ samples are maintained uniformly from the stream, each incoming example has the same probability of being stored, leading to:

$$\mathscr{L}_{n_c} + \alpha \mathbb{E}_{(x_{\mathscr{B}},z)\sim\mathscr{B}}\Big[D_{KL}\big(\text{softmax}(z)\|f_\theta(x_{\mathscr{B}})\big)\Big]. \tag{3.8}$$

Under mild assumptions [20], minimizing the KL divergence between softened outputs is equivalent to minimizing the *Euclidean distance* between logits (in practice, expecta-

Algorithm 2 DER algorithm

Input: (x_n, y_n) current mini-batch at step n; replay buffer $\mathscr{B}$ with capacity $|\mathscr{B}|$; h_θ logits of the model; $(x_{\mathscr{B}}, z) \in \mathscr{B}$ content of the buffer; distillation weight α.
$\mathscr{B} \leftarrow \emptyset$
for $n \in \{1, \dots, N\}$ **do**
    Training
    $\mathscr{L} \leftarrow \mathscr{L}_{CE}\Big(f_\theta\big(x_n\big), y_n\Big)$ ▷ CE on new samples
    $\mathscr{L} \leftarrow \mathscr{L} + \alpha\, \mathscr{L}_{MSE}\Big(z, h_\theta(x_{\mathscr{B}})\Big)$ ▷ Logits distillation via MSE
    Update buffer
    $x_{\mathscr{B}} \leftarrow x_n$ ▷ Update examples
    $\mathscr{B} \leftarrow \big(h_\theta(x_{\mathscr{B}})\big)$ ▷ Update logits
end for

tions are approximated through mini-batch gradients). Therefore, **DER** optimizes:

$$\mathscr{L}_{n_c} + \alpha \mathbb{E}_{(x_{\mathscr{B}}, z) \sim \mathscr{B}}\Big[\|z - h_\theta(x_{\mathscr{B}})\|_2^2\Big]. \tag{3.9}$$

DER is illustrated in Figure 3.2 and detailed in Algorithm 2.

3.1.4 Dark Experience Replay++

A limitation of **DER** emerges when the input stream undergoes an abrupt distribution shift. In this case, logits stored in the buffer may be strongly biased by the training state in which they were produced, and replaying only those logits can reduce effectiveness. A simple way to mitigate this issue is to also retain the *ground-truth labels* y.

DER++ augments the **DER** objective with an additional supervised term computed on replayed samples, encouraging a higher conditional likelihood with respect to their labels while introducing only a small memory overhead:

$$\mathscr{L}_{t_c} + \alpha \mathbb{E}_{(x', y', z') \sim \mathscr{B}}\Big[\|z' - h_\theta(x')\|_2^2\Big] + \beta\, \mathbb{E}_{(x'', y'', z'') \sim \mathscr{B}}\Big[\ell\big(y'', f_\theta(x'')\big)\Big], \tag{3.10}$$

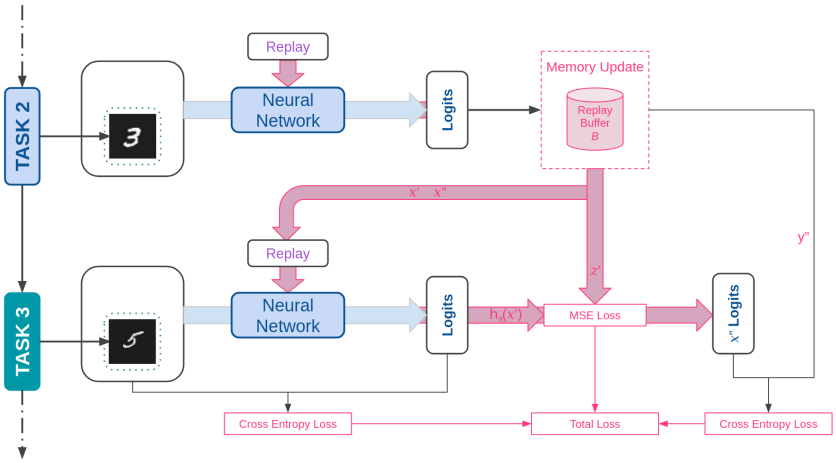

Figure 3.3: Overview of DER++ method. The replay buffer stores past samples together with their logits and labels. When learning a new task, buffered samples contribute an MSE loss that matches current logits to stored logits and an additional cross-entropy loss on replayed labels, which are combined with the current-task loss to better preserve prior knowledge and reduce catastrophic forgetting.

where β is an additional hyperparameter controlling the contribution of the last term (and **DER++** reduces to **DER** when $\beta = 0$).

DER++ is illustrated in Figure 3.3 and detailed in Algorithm 3.

3.2 Soft-DTW Divergence

In many time-series learning problems, a point-wise comparison between two sequences is often too restrictive. Two signals may encode the same underlying pattern while being locally shifted, stretched, or compressed along the time axis. In such cases, a useful similarity measure should tolerate temporal misalignment while still being compatible with gradient-based optimization.

For this reason, we first review *Dynamic Time Warping* (**DTW**), a classical dynamic-

Algorithm 3 DER++ algorithm

Input: (x_n, y_n) current mini-batch at step n; replay buffer $\mathscr{B}$ with capacity $|\mathscr{B}|$; h_θ logits of the model; $(x', y', z', x'', y'', z'') \in \mathscr{B}$ content of the buffer; distillation weight α; buffer cross-entropy weight β.
$\mathscr{B} \leftarrow \emptyset$
for $n \in \{1, \dots, N\}$ **do**
    Training
    $\mathscr{L} \leftarrow \mathscr{L}_{CE}\Big(f_\theta\big(x_n\big), y_n\Big)$ ▷ CE on new samples
    $\mathscr{L} \leftarrow \mathscr{L} + \alpha\, \mathscr{L}_{MSE}\Big(z', h_\theta(x')\Big)$ ▷ Logits distillation via MSE
    $\mathscr{L} \leftarrow \mathscr{L} + \beta\, \mathscr{L}_{CE}\Big(f_\theta\big(x''\big), y''\Big)$ ▷ CE on buffer samples
    Update buffer
    $x' \leftarrow x_n, \quad x'' \leftarrow x_n, \quad y'' \leftarrow y_n$ ▷ Update examples
    $\mathscr{B} \leftarrow \big(h_\theta(x')\big)$ ▷ Update logits
end for

programming technique for aligning temporal sequences under monotonicity and continuity constraints. We then introduce *Soft-DTW*, a differentiable relaxation of **DTW** in which the hard minimum over alignment costs is replaced by a smooth *soft-minimum*, making the objective suitable for Deep Learning pipelines. Finally, we present the *Soft-DTW divergence*, which removes the entropic bias of *Soft-DTW* and yields a proper non-negative divergence. This property is particularly useful when alignment-based losses are used as optimization terms.

3.2.1 Dynamic Time Warping

The problem of aligning *time series* appears in a wide range of applications. The goal is to match two sequences that share the same global structure but differ locally in timing (e.g., local accelerations, slow-downs, or phase shifts). In this context, the quality of the alignment strongly depends on the *local similarity* (or cost) used to compare samples,

especially in high-dimensional settings.

Dynamic Time Warping (**DTW**) is a classical *dynamic-programming* method that finds an optimal monotonic alignment between two sequences by accumulating local matching costs under continuity and ordering constraints [6]. **DTW** is effective in the presence of temporal distortions, but its performance depends on the choice of the local cost used to build the pairwise cost matrix. For this reason, it is convenient to formulate **DTW** using a generic cost matrix, so that both standard distances and learned similarities can be handled within the same framework.

Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ and $\mathbf{Y} \in \mathbb{R}^{m \times d}$ be two d-dimensional time series of lengths n and m, respectively. Their samples are denoted by $\mathbf{x}_i \in \mathbb{R}^d$ for $i \in [n]$ and $\mathbf{y}_j \in \mathbb{R}^d$ for $j \in [m]$. Let $\Delta(\mathbf{X}, \mathbf{Y}) \in \mathbb{R}^{n \times m}$ be the pairwise cost matrix, where $\Delta(\mathbf{X}, \mathbf{Y})_{i,j}$ is the cost of aligning $\mathbf{x}_i$ with $\mathbf{y}_j$. A common choice is the squared Euclidean cost:

$$\Delta(\mathbf{X}, \mathbf{Y})_{i,j} = \|\mathbf{x}_i - \mathbf{y}_j\|_2^2. \tag{3.11}$$

An alignment between $\mathbf{X}$ and $\mathbf{Y}$ can be represented by a binary matrix $A \in \{0,1\}^{n \times m}$, where $A_{i,j} = 1$ if the samples $\mathbf{x}_i$ and $\mathbf{y}_j$ are matched, and $A_{i,j} = 0$ otherwise. We say that A is a *monotonic alignment matrix* if its non-zero entries form a path from the upper-left corner $(1,1)$ to the lower-right corner (n,m) using only the moves $\downarrow$, $\rightarrow$, and $\searrow$. The set of all such valid alignments is denoted by

$$\mathscr{A}(n,m) \subset \{0,1\}^{n \times m}.$$

Figure 3.4 shows two possible alignments between two time series.

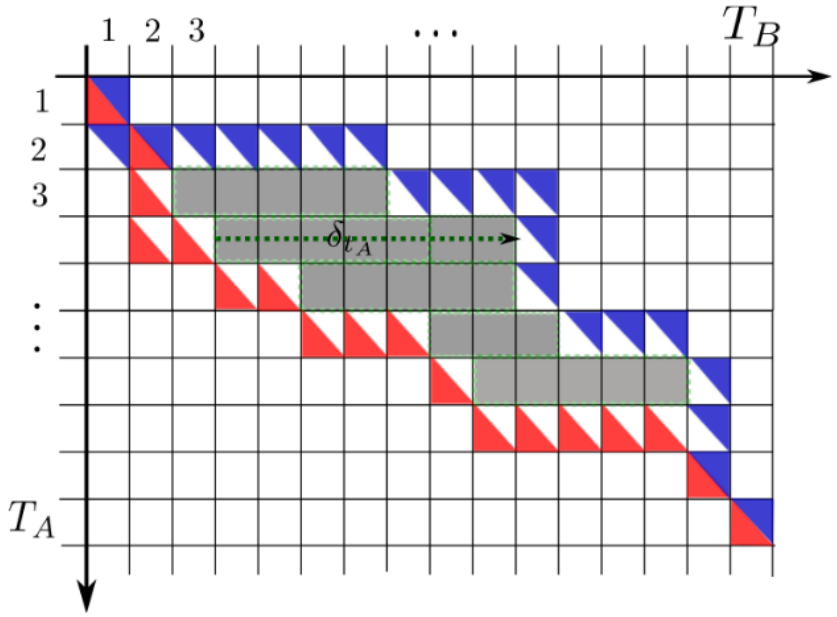

Figure 3.4: Example of two valid alignments represented by matrices Y^1 (red upper triangles) and Y^2 (blue lower triangles). The gray zone corresponds to the area loss δ_{t_A} between Y^1 and Y^2. Figure by courtesy of [6].

Given a cost matrix $\Delta(\mathbf{X},\mathbf{Y})$, the cost of an alignment $A \in \mathcal{A}(n,m)$ is given by the Frobenius inner product

$$\langle A, \Delta(\mathbf{X},\mathbf{Y})\rangle = \sum_{i=1}^{n}\sum_{j=1}^{m} A_{i,j}\,\Delta(\mathbf{X},\mathbf{Y})_{i,j}. \tag{3.12}$$

DTW selects the alignment with minimum cumulative cost:

$$\mathrm{DTW}(\mathbf{X},\mathbf{Y}) = \min_{A\in\mathcal{A}(n,m)} \langle A, \Delta(\mathbf{X},\mathbf{Y})\rangle. \tag{3.13}$$

This formulation is equivalent to the affinity-maximization view often used in the literature (by taking an affinity matrix as the negative of the cost matrix). More expressive formulations may also replace the Euclidean cost with parameterized or *learned pairwise costs*, which can improve robustness and discriminability.

3.2.2 Soft-DTW and Soft-DTW Divergence

Many time-indexed signals can undergo local stretching or compression without changing their semantic content. In addition, different acquisition processes may produce sequences with different lengths or asynchronous sampling. For this reason, generative and predictive models for time series often benefit from objectives that are invariant (or at least robust) to temporal misalignment. Although **DTW** naturally handles temporal shifts and local dilations by searching for an optimal alignment path, it is *not differentiable* and can be *unstable* when directly used inside gradient-based optimization procedures. This limits its applicability as a loss function in deep learning models. *Soft-DTW* addresses this limitation by replacing the hard minimum over alignment costs with a *soft-minimum*, resulting in a smooth version of **DTW** that remains fully differentiable [7]. This makes it suitable as an end-to-end loss for neural architectures (e.g., **MLPs** or **RNNs**) in both predictive and generative time-series tasks.

As in the previous subsection, let $\mathbf{X} \in \mathbb{R}^{n \times d}$ and $\mathbf{Y} \in \mathbb{R}^{m \times d}$ be two multivariate discrete time series, and let $\Delta(\mathbf{X}, \mathbf{Y}) \in \mathbb{R}^{n \times m}$ be a differentiable cost matrix (typically induced by a point-wise cost such as Euclidean distance). The score of a valid alignment $A \in \mathscr{A}(n, m)$ is again $\langle A, \Delta(\mathbf{X}, \mathbf{Y}) \rangle$. Figure 3.5 shows the cost associated with multiple alignment paths. The generalized soft-minimum operator with smoothing parameter $\gamma \geq 0$ is defined as:

$$\min^{\gamma}\{a_1, \ldots, a_N\} := \begin{cases} \min_{1 \leq i \leq N} a_i, & \gamma = 0, \\ -\gamma \log \sum_{i=1}^{N} e^{-a_i/\gamma}, & \gamma > 0. \end{cases} \tag{3.14}$$

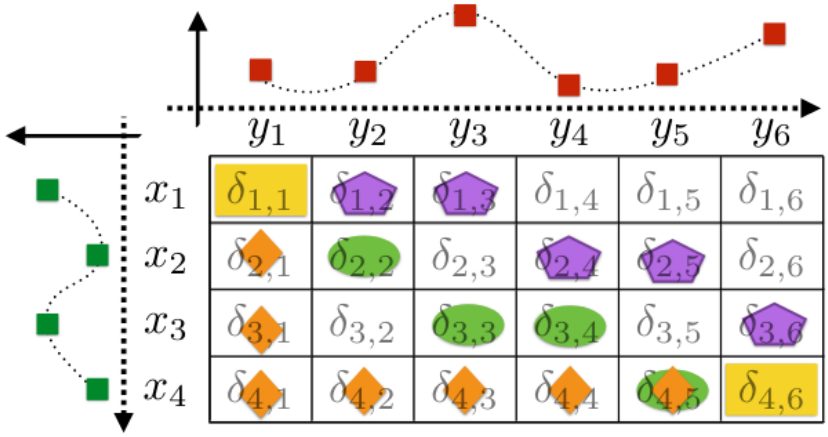

Figure 3.5: Three alignment matrices (orange, green, purple, in addition to the top-left and bottom-right entries) between two time series of length 4 and 6. The cost of an alignment is equal to the sum of entries visited along the path. Soft-DTW considers all possible alignment matrices. Figure by courtesy of [7].

Using this operator, we can declare the γ-Soft-DTW between $\mathbf{X}$ and $\mathbf{Y}$:

$$\mathrm{SDTW}_\gamma(\mathbf{X},\mathbf{Y}) := \min^\gamma \left\{ \langle A, \Delta(\mathbf{X},\mathbf{Y}) \rangle \ : \ A \in \mathscr{A}(n,m) \right\}. \tag{3.15}$$

The original **DTW** is recovered in the limit $\gamma \to 0$. On the other hand, as γ increases, the objective increasingly aggregates information from many alignment paths rather than focusing only on the single best one. In practice, this smoothing improves the *optimization landscape* and makes gradient-based training more stable, at the cost of introducing a *controlled bias* with respect to hard **DTW**.

Despite these advantages, *Soft-DTW* is not, in general, a proper divergence. In particular, when $\gamma > 0$, the *entropic smoothing* induces a bias that can make the self-similarity term non-zero (and the value can even become negative, depending on the chosen ground cost). As a consequence, minimizing raw *Soft-DTW* does not always guarantee

that identical sequences achieve the minimum value.

To correct this issue, *Soft-DTW divergence* subtracts the self-similarity terms and removes the entropic bias [8]. Let $\mathscr{C}$ denote a (possibly parameterized) cost-matrix mapping $\mathscr{C} : \mathbb{R}^{n\times d} \times \mathbb{R}^{m\times d} \to \mathbb{R}^{n\times m}$, with $\mathscr{C}(\mathbf{X},\mathbf{Y})$ typically instantiated as a squared Euclidean cost matrix. Then the *Soft-DTW divergence* is defined as:

$$D_\gamma^{\mathscr{C}}(\mathbf{X},\mathbf{Y}) := \mathrm{SDTW}_\gamma\big(\mathscr{C}(\mathbf{X},\mathbf{Y})\big) - \frac{1}{2}\mathrm{SDTW}_\gamma\big(\mathscr{C}(\mathbf{X},\mathbf{X})\big) - \frac{1}{2}\mathrm{SDTW}_\gamma\big(\mathscr{C}(\mathbf{Y},\mathbf{Y})\big). \tag{3.16}$$

Equivalently, when the cost matrix is understood from context, we may write $D_\gamma(\mathbf{X},\mathbf{Y})$ for brevity.

Under suitable assumptions on the ground cost, $D_\gamma^{\mathscr{C}}$ is a valid divergence: it is non-negative,

$$D_\gamma^{\mathscr{C}}(\mathbf{X},\mathbf{Y}) \geq 0 \qquad \forall \mathbf{X},\mathbf{Y},$$

and it satisfies the identity of indiscernibles,

$$D_\gamma^{\mathscr{C}}(\mathbf{X},\mathbf{Y}) = 0 \iff \mathbf{X} = \mathbf{Y}.$$

These properties make *Soft-DTW divergence* especially appealing in learning objectives where temporal alignment is required but a well-behaved optimization target is also needed.

Chapter 4

Models

This chapter presents the architectures considered in this thesis and clarifies how the proposed method is instantiated in practice. We first introduce the common residual backbone adopted across all experiments, since architectural parity is necessary to compare **ANN**-based and **SNN**-based Continual Learning methods in a controlled setting. We then describe the replay-based baselines used as references, namely **ER**, **DER**, and **DER++**, together with their spiking counterparts. Finally, we detail **STAER**, whose central idea is to preserve not only the semantic content of past examples, but also the temporal evolution of the corresponding spiking responses.

4.1 Backbone

To ensure a fair comparison across paradigms, all methods considered in this thesis are implemented on the same residual architecture, differing only in the neuron model and in the learning objective. In particular, both the standard **ANN** models and the

spiking variants are built on a *ResNet19* backbone, following recent directly-trained **SNN** literature [21–24]. This design choice removes a major source of variability from the comparison: performance differences can then be attributed primarily to the *Continual Learning strategy* and to the *temporal modeling capability* of the network, rather than to changes in representational capacity.

The adopted backbone consists of an initial convolutional stem with 128 output channels, followed by three residual stages composed of 3, 3, and 2 Basic Blocks, respectively. The channel widths of the three stages are $\{128, 256, 512\}$, and spatial downsampling is performed at the beginning of the last two stages. The classifier head is formed by global average pooling followed by a linear layer that maps the final representation to the output space.

The **ANN** instantiation follows the standard residual design, where each block uses convolutional layers, Batch Normalization, and ReLU activations. The **SNN** instantiation preserves the same spatial structure, but replaces conventional activations with *Leaky Integrate-and-Fire* (**LIF**) neurons and uses normalization compatible with the temporal tensor representation. In our implementation, the spiking backbone uses membrane decay factor $\gamma = 0.5$, firing threshold $V_{\text{th}} = 1.0$, and is trained with *ArcTan surrogate gradient*, as discussed in Section 2.4.2.

A relevant difference between the two instantiations concerns the output. The **ANN** produces a single logit vector for each input image, whereas the **SNN** produces a sequence of logits over the temporal dimension. Let $h^T(x) \in \mathbb{R}^{T \times C}$ denote the temporal logits associated with an input x, where T is the number of time steps and C is the

number of classes. The time-aggregated prediction used for classification is obtained by averaging logits over the temporal axis:

$$\bar{h}(x) = \frac{1}{T} \sum_{t=1}^{T} h^T(x)[t]. \tag{4.1}$$

This formulation is particularly convenient in our setting. First, it preserves a dense real-valued supervision signal, which is easier to optimize than a pure spike-count objective. Second, it leaves the full temporal trajectory available, which is essential for the alignment mechanism introduced by **STAER**. Finally, because the classifier head does not depend on a fixed temporal dimension, the same trained **SNN** can be evaluated under different temporal resolutions, a property that is directly exploited by the contraction/expansion mechanism of the proposed method.

4.2 Replay-based Baselines

We compare **STAER** against three replay-based **CL** baselines that are widely used in the **ANN** literature, introduced in Section 3.1: **ER**, **DER**, and **DER++**. These methods are simple, strong, and particularly suitable as references because they differ mainly in what is stored in memory and in how replayed samples contribute to the loss.

In their original formulation, these baselines are implemented with conventional **ANN** backbones. In this thesis, we also consider their spiking counterparts, denoted as *snn-ER*, *snn-DER*, and *snn-DER++*. To preserve comparability, the original *ResNet18* backbone used in prior work is replaced with the same *ResNet19* architecture adopted by our method. The replay logic remains unchanged: *snn-ER* stores input-label pairs, whereas *snn-DER* and *snn-DER++* additionally exploit buffered outputs of the spiking

classifier. In all cases, classification and replay losses are applied to the real-valued logit outputs of the network, not to spike counts. As a result, the comparison focuses on the *Continual Learning objective* rather than on differences in model size or classifier head design.

4.3 STAER

The proposed method, **S**piking **T**emporal **A**lignment with **E**xperience **R**eplay (**STAER**) [2], starts from a simple observation: in a **SNN**, forgetting may appear not only as a change in the final class decision, but also as a drift in the temporal structure of the response. Standard replay is useful because it exposes the model to past examples, but it does not explicitly constrain *when* information is produced over time. **STAER** addresses this point by coupling replay with a temporal-alignment loss defined on the sequence of logits produced by the spiking network.

4.3.1 Spiking Neuron Model

Following recent directly-trained **SNN** works [21,24], each spiking layer is modeled with discrete-time **LIF** neurons. For a given layer ℓ and time step t, the synaptic current $\mathbf{I}^\ell[t]$ and the membrane potential $\mathbf{u}^\ell[t]$ evolve as

$$\mathbf{I}^\ell[t] = \mathbf{W}^\ell\,\mathbf{s}^{\ell-1}[t], \qquad \mathbf{u}^\ell[t] = \gamma\,\mathbf{u}^\ell[t-1] + \mathbf{I}^\ell[t] - \mathbf{s}^\ell[t-1]V_{\text{th}}, \tag{4.2}$$

where $\mathbf{W}^\ell$ denotes the learnable weight matrix, $\mathbf{s}^\ell[t] \in \{0,1\}^{m_\ell}$ is the spike vector of layer ℓ, m_ℓ is the number of neurons in that layer, $\gamma \in (0,1)$ is the leak coefficient, and V_{th} is

the firing threshold. Spike emission is determined by the Heaviside firing function

$$\mathbf{s}^{\ell}[t] = H\big(\mathbf{u}^{\ell}[t] - V_{\text{th}}\big). \tag{4.3}$$

The subtractive term $\mathbf{s}^{\ell}[t-1]V_{\text{th}}$ implements a soft reset after spike emission.

Since $H(\cdot)$ is non-differentiable, optimization is performed with surrogate gradients. As in the implementation used by *snnTorch*, the forward pass preserves the hard thresholding behavior, while the backward pass replaces the derivative with a smooth proxy [25]. We adopt the ArcTan surrogate (introduced in Section 2.4.2):

$$\frac{\partial \mathbf{s}}{\partial \mathbf{u}} \approx g(\mathbf{u} - V_{\text{th}}), \tag{4.4}$$

with

$$g(z) = \frac{\alpha}{2} \frac{1}{1 + \left(\frac{\pi\alpha}{2} z\right)^2},$$

where α controls the slope around threshold.

4.3.2 Replay with Multi-scale Temporal Traces

Unlike standard spiking classifiers that use spike counts as the final decision signal, **STAER** produces real-valued logits at every time step, following the strategy introduced in [24]. This choice is central to the method, because it makes temporal responses directly comparable across tasks. For a replay sample x, the buffer stores not only the pair (x, y), but also three temporal logit sequences corresponding to three temporal resolutions:

$$h_{\mathscr{B}}^{T/2}, \qquad h_{\mathscr{B}}^{T}, \qquad h_{\mathscr{B}}^{2T}.$$

Hence, each memory element can be viewed as a tuple

$$(x, y, h_{\mathscr{B}}^{T/2}, h_{\mathscr{B}}^{T}, h_{\mathscr{B}}^{2T}).$$

The three stored traces are obtained through the *static encoding* adopted in the experimental setup (later detailed in Section 5.1). A static input image is replicated along the temporal dimension to form a sequence of length T. Temporal contraction and expansion are then obtained simply by changing the replication length to $T/2$ or $2T$. Since the backbone is not tied to a single fixed temporal dimension, the same trained model can process all three variants without changing the parameters. Conceptually, this mechanism exposes the replay process to compressed and dilated versions of the same response, which is consistent with the biological intuition that memory recall may undergo temporal compression or expansion [26, 27].

The role of this design is easier to appreciate by contrasting it with the baselines. **ER** preserves input-label pairs, and **DER**/**DER++** additionally preserve static prediction targets. **STAER**, instead, preserves an entire *temporal trajectory*. In other words, the method does not only ask the current network to classify an old sample correctly; it also asks the network to recover a compatible temporal evolution of the logits associated with that sample.

4.3.3 Training Objective

Let (x_n, y_n) denote the current-task mini-batch at task n, and let $(x_{\mathscr{B}}, y_{\mathscr{B}})$ denote a mini-batch sampled from the replay buffer $\mathscr{B}$. The supervised term is computed on

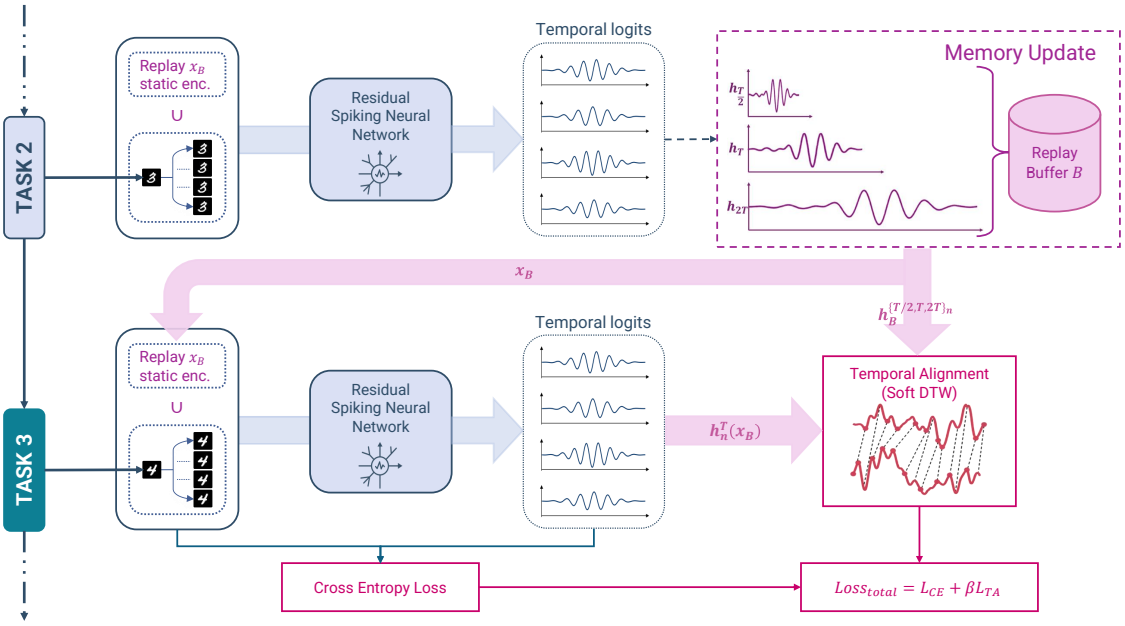

Figure 4.1: For each input, logits are stored at three temporal resolutions $(T, T/2, 2T)$ in the replay buffer to mimic biological memory variability. The training objective combines cross-entropy on current task samples with a *Temporal Alignment* loss based on Soft-DTW divergence [8], computed between current and past logits at multiple temporal scales. Reproduced from [2].

the concatenation of current and replayed samples:

$$x = [x_n; x_{\mathcal{B}}], \qquad y = [y_n; y_{\mathcal{B}}]. \tag{4.5}$$

Using temporally averaged logits,

$$\bar{h}_n(x) = \frac{1}{T}\sum_{t=1}^{T} h_n^T(x)[t], \tag{4.6}$$

we define the cross-entropy loss as

$$\mathscr{L}_{CE} = \mathrm{CE}\big(\bar{h}_n(x), y\big). \tag{4.7}$$

This term plays the same role as in standard replay-based methods: it preserves discriminative performance on both new and buffered classes.

The distinctive part of **STAER** is the *Temporal Alignment* term $\mathscr{L}_{TA}$. For each replayed sample, the current model produces temporal logits $h_n^T(x_{\mathscr{B}}) \in \mathbb{R}^{T\times C}$. These logits are aligned against the three temporal traces stored in the buffer using the *Soft-DTW divergence* introduced in Section 3.2.2. This yields three alignment terms:

$$\mathscr{L}^{T}_{\mathrm{SDTW}}(x_{\mathscr{B}}) = \mathscr{L}_{\mathrm{SDTW}}\Big(h_n^T(x_{\mathscr{B}}),\, h^T_{\mathscr{B}}\Big), \tag{4.8}$$

$$\mathscr{L}^{T/2}_{\mathrm{SDTW}}(x_{\mathscr{B}}) = \mathscr{L}_{\mathrm{SDTW}}\Big(h_n^T(x_{\mathscr{B}}),\, h^{T/2}_{\mathscr{B}}\Big), \tag{4.9}$$

$$\mathscr{L}^{2T}_{\mathrm{SDTW}}(x_{\mathscr{B}}) = \mathscr{L}_{\mathrm{SDTW}}\Big(h_n^T(x_{\mathscr{B}}),\, h^{2T}_{\mathscr{B}}\Big). \tag{4.10}$$

The overall temporal-alignment loss is a normalized weighted combination of the three terms:

$$\mathscr{L}_{TA} = \frac{\mathscr{L}^{T}_{\mathrm{SDTW}} + \alpha_1\, \mathscr{L}^{T/2}_{\mathrm{SDTW}} + \alpha_2\, \mathscr{L}^{2T}_{\mathrm{SDTW}}}{1+\alpha_1+\alpha_2}, \tag{4.11}$$

where α_1 and α_2 control the relative contribution of temporal contraction and temporal expansion.

The final objective is therefore:

$$\mathscr{L} = \mathscr{L}_{CE} + \beta\, \mathscr{L}_{TA}, \tag{4.12}$$

where β balances current-task supervision and temporal preservation. The interpretation of this objective is straightforward. The term $\mathscr{L}_{CE}$ ensures that the model remains a good classifier over the growing label space, while $\mathscr{L}_{TA}$ regularizes the *shape* of past responses over time. The method therefore acts simultaneously on two levels: it preserves *what* the network predicts and, crucially for spiking models, also *how* those predictions unfold along the temporal axis.

At the end of each update step, selected current-task samples are inserted into the replay memory together with the three temporal traces produced by the current model.

Algorithm 4 STAER algorithm

Input: $(x_n, y_n) \in \mathscr{D}_n$ new task-n data; $h_n^T(x)$ logits of the model trained up to task n with time window T; $(x_{\mathscr{B}}, y_{\mathscr{B}}, h_{\mathscr{B}}^{T/2}, h_{\mathscr{B}}^{T}, h_{\mathscr{B}}^{2T}) \in \mathscr{B}$ buffer contents.
$\mathscr{B} \leftarrow \emptyset$
for $n \in \{1, \ldots, N\}$ **do**
    Training
    $\mathscr{L} \leftarrow \mathscr{L}_{CE}\Big(\bar{h}_n\big([x_n, x_{\mathscr{B}}]\big), [y_n, y_{\mathscr{B}}]\Big)$ ▷ CE on new+buffer
    $\mathscr{L} \leftarrow \mathscr{L} + \beta\, \mathscr{L}_{TA}\Big(h_n^T(x_{\mathscr{B}}), h_{\mathscr{B}}^{T/2}, h_{\mathscr{B}}^{T}, h_{\mathscr{B}}^{2T}\Big)$ ▷ Temporal alignment
    Update buffer
    $x_{\mathscr{B}} \leftarrow x_n, \quad y_{\mathscr{B}} \leftarrow y_n$ ▷ Update examples
    $\mathscr{B} \leftarrow (h_n^{T/2}(x_{\mathscr{B}}),\ h_n^T(x_{\mathscr{B}}),\ h_n^{2T}(x_{\mathscr{B}}))$ ▷ Update multi-time logits
end for

In this way, the buffer continuously stores a compressed record of past experience that includes both semantic supervision and multi-scale temporal information. The complete procedure is summarized in Algorithm 4.

Chapter 5

Empirical Study

In this chapter, we conduct a systematic empirical evaluation of **STAER** in the Continual Learning setting, where sequential exposure to tasks induces *catastrophic forgetting*. We begin by introducing the two standard benchmarks considered in our study—Sequential-MNIST and Sequential-CIFAR10—and by detailing the input encoding adopted to enable a principled comparison between spiking and non-spiking paradigms. We then describe the experimental protocol used throughout, including the **CL** scenarios under which models are assessed.

To contextualize the impact of our contributions, we compare **STAER** against established replay-based baselines—**ER**, **DER**, and **DER++**—both in their original **ANN** version and converted **SNN** one, as well as two reference bounds: sequential fine-tuning via *SGD* as a lower baseline and *Joint training* over all tasks as an upper bound. Performance is reported under both *Task-Incremental* and *Class-Incremental* Learning protocols, with particular emphasis on the latter, where task identity is not available

at inference and forgetting is typically most severe. Finally, we provide an ablation of the proposed *Temporal Alignment objective* $\mathscr{L}_{TA}$, identifying the regimes in which alignment is beneficial and motivating the hyperparameter choices used in the main experiments, before concluding with a comprehensive discussion of results in terms of Final Average Accuracy and Forgetting across datasets, buffer sizes, and temporal resolutions.

5.1 Datasets

We evaluate **STAER** on two standard Continual Learning benchmarks, Sequential-MNIST and Sequential-CIFAR10 [28]. Each benchmark contains 10 classes, partitioned into 5 sequential tasks comprising 2 novel classes per task.

Since **STAER** operates on temporally extended inputs, whereas the **ANN** baselines process static images, we adopt a *static encoding* procedure that transforms each image into a length-T sequence by repeating the same frame across T time steps. For a mini-batch of size B, this yields an input tensor of shape $T \times B \times C \times H \times W$, where C denotes the number of channels and $H \times W$ the spatial resolution. Within this formulation, temporal contraction and expansion are implemented by modifying the replication factor in the encoding, producing sequences of length $T/2$ and $2T$, respectively.

Sequential-MNIST consists of 28×28 grayscale digit images. Sequential-CIFAR10 comprises 32×32 RGB images and is trained with standard data augmentation, including random cropping, horizontal flipping, and normalization.

5.2 Experimental settings

We compare **STAER** against standard replay-based continual-learning approaches and against two reference training regimes, considering both **ANN** and **SNN** implementations. As a lower reference, we report the performance of plain sequential optimization with *SGD*, i.e., fine-tuning across tasks without any specific mechanism to mitigate forgetting. As an upper reference, we consider *Joint training*, where the model is optimized on the union of all tasks simultaneously.

In line with the standard **CL** literature, we conduct the evaluation under protocols that differ in the information available at inference time and in the way the task stream is defined. Following the scenarios introduced in Section 2.3.2, we consider both *Task-Incremental Learning* (**TIL**) and *Class-Incremental Learning* (**CIL**) [1]. The **TIL** setting is generally less challenging, since the task label is known during testing. Our primary interest, however, is in **CIL**, where tasks are presented sequentially over disjoint subsets of classes and the model must predict among all classes encountered so far, without being informed of the task identity at test time.

Table 5.1 summarizes the results using *Final Average Accuracy* (**FAA**), namely the mean classification accuracy over all tasks after the completion of the full incremental training sequence. Denoting by $R_{N,n}$ the accuracy on task n measured after training on the final task N, **FAA** is defined as:

$$\mathrm{FAA} = \frac{1}{N}\sum_{n=1}^{N} R_{N,n}. \tag{5.1}$$

While informative, **FAA** alone does not directly capture the extent to which previously

Table 5.1: Results on Sequential-MNIST and Sequential-CIFAR10 under CIL. Bold and underlined values denote the best and second-best results within each SNN block (T=2 and T=4), respectively. Reproduced from [2].

Method	**S-MNIST**			**S-CIFAR-10**		
Buffer size	*200*	*500*	*5120*	*200*	*500*	*5120*
ANN without T						
JOINT		99.32			98.25	
SGD		19.89			19.63	
ER	94.46	95.39	98.17	46.68	61.64	85.91
DER	96.12	97.91	98.83	51.71	64.51	86.27
DER++	96.37	98.15	99.06	56.57	65.94	87.08
SNN with T=2						
snn-JOINT		98.72			89.75	
snn-SGD		19.64			19.44	
snn-ER	87.90	94.06	95.20	37.80	52.68	72.89
snn-DER	92.54	94.88	96.88	<u>48.56</u>	58.62	75.92
snn-DER++	<u>93.54</u>	<u>96.01</u>	<u>97.33</u>	**50.40**	<u>59.08</u>	<u>78.19</u>
STAER	**93.71**	**96.70**	**97.53**	47.88	**59.12**	**78.30**
SNN with T=4						
snn-JOINT		98.92			89.80	
snn-SGD		19.68			19.47	
snn-ER	91.20	94.00	96.55	38.99	50.82	74.44
snn-DER	<u>94.07</u>	95.79	97.78	50.16	57.24	75.80
snn-DER++	93.92	<u>96.00</u>	<u>98.04</u>	<u>51.11</u>	<u>60.81</u>	<u>78.25</u>
STAER	**94.18**	**96.92**	**98.48**	**51.19**	**65.68**	**83.53**

acquired knowledge deteriorates over time. For this reason, we also report in Table 5.2 the *Forgetting* metric (**FRG**), computed from the accuracy matrix $\{R_{k,n}\}$, where $R_{k,n}$ denotes the accuracy on task n after learning up to task k. For each task $n < N$,

Table 5.2: Forgetting results on Sequential-MNIST and Sequential-CIFAR10 under CIL. Bold and underlined values denote the best and second-best results within each SNN block (T=2 and T=4), respectively. Reproduced from [2].

FORGETTING						
Method	**S-MNIST**			**S-CIFAR-10**		
Buffer size	*200*	*500*	*5120*	*200*	*500*	*5120*
ANN without T						
ER	6.10	5.33	1.46	63.11	44.66	17.15
DER	4.46	1.57	0.66	55.85	44.23	13.47
DER++	4.04	1.94	0.04	47.81	36.41	11.60
SNN with T=2						
snn-ER	14.40	6.03	4.88	71.72	55.52	19.83
snn-DER	6.97	2.27	1.77	47.80	35.38	18.78
snn-DER++	5.07	1.92	1.05	42.41	**31.86**	15.61
STAER	**4.05**	**1.70**	**0.67**	**41.91**	33.68	**13.85**
SNN with T=4						
snn-ER	10.16	4.23	1.82	70.31	53.2	19.37
snn-DER	4.82	2.08	0.92	38.55	31.83	13.39
snn-DER++	4.27	1.44	0.80	**32.66**	31.42	13.46
STAER	**2.14**	**0.94**	**0.44**	35.47	**29.20**	**11.45**

forgetting is measured as the gap between the best accuracy ever achieved on that task and the corresponding accuracy after the final task. The aggregated measure is:

$$\mathrm{FRG} = \frac{1}{N-1}\sum_{n=1}^{N-1}\Big(\max_{k\in\{n,\dots,N\}} R_{k,n} - R_{N,n}\Big), \tag{5.2}$$

with smaller values indicating stronger retention of previous knowledge.

In the *Temporal-Alignment objective* $\mathscr{L}_{TA}$, the contraction and expansion components, $\mathscr{L}_{\mathrm{SDTW}}^{T/2}$ and $\mathscr{L}_{\mathrm{SDTW}}^{2T}$, are evaluated on half of each mini-batch. We consider three replay-

memory capacities, namely $\mathscr{B} \in \{200, 500, 5120\}$, consistent with common practice in the literature [19]. On Sequential-CIFAR10, all models are trained for 50 epochs per task with batch size 32, whereas on Sequential-MNIST we train for 1 epoch per task with batch size 10. The temporal dimension is set to $T \in \{2, 4\}$. Unless otherwise specified, we use $\alpha_1 = \alpha_2 = 0.5$ and $\beta = 10^{-4}$. Optimization is performed with Adam [29], using a learning rate of 3×10^{-3} together with a cosine annealing schedule.

5.3 Results

The results reported in Table 5.1 show that, under the **CIL** setting, **STAER** consistently improves over the spiking replay-based baselines on both benchmarks. In particular, when $T = 4$, our method achieves the best **SNN** performance across all replay-buffer sizes on both Sequential-MNIST and Sequential-CIFAR10, reaching up to 98.48% and 83.53%, respectively. In this regime, **STAER** systematically surpasses *snn-ER* and also provides clear improvements over *snn-DER* and *snn-DER++*. When the temporal dimension is reduced to $T = 2$, our method remains competitive on Sequential-MNIST and still improves upon *snn-ER*, whereas on the more demanding Sequential-CIFAR10 benchmark the advantage becomes less uniform, especially for larger buffers. Overall, the contribution of the *Temporal Alignment* objective is more evident when the number of time steps increases, with the largest gains emerging on Sequential-CIFAR10 (i.e. +6.56% at B=500). This suggests that the proposed objective benefits from a richer temporal representation, while the competing spiking replay methods are comparatively less sensitive to an increase in temporal resolution.

Table 5.3: Results on Sequential-MNIST and Sequential-CIFAR10 under TIL. Bold and underlined values denote the best and second-best results within each SNN block (T=2 and T=4), respectively. Reproduced from [2].

Method	**S-MNIST**			**S-CIFAR-10**		
Buffer size	*200*	*500*	*5120*	*200*	*500*	*5120*
ANN without T						
JOINT		99.99			98.25	
SGD		51.71			55.15	
ER	99.52	99.73	99.93	88.32	91.68	93.62
DER	99.65	99.82	99.91	90.21	92.24	96.20
DER++	99.67	99.83	99.93	91.24	93.67	97.19
SNN with T=2						
snn-JOINT		99.72			96.05	
snn-SGD		57.55			59.37	
snn-ER	98.65	99.31	99.20	83.17	83.99	90.51
snn-DER	**99.28**	99.26	99.55	83.49	86.57	92.70
snn-DER++	99.07	<u>99.62</u>	<u>99.70</u>	84.57	**89.38**	<u>94.24</u>
STAER	<u>99.17</u>	**99.63**	**99.71**	**84.62**	<u>88.93</u>	**94.28**
SNN with T=4						
snn-JOINT		99.92			96.75	
snn-SGD		63.85			62.67	
snn-ER	98.89	99.42	99.73	83.91	84.23	90.92
snn-DER	99.33	99.35	99.75	85.50	88.01	93.39
snn-DER++	<u>99.43</u>	<u>99.74</u>	<u>99.77</u>	<u>86.93</u>	<u>90.22</u>	<u>95.03</u>
STAER	**99.45**	**99.81**	**99.88**	**88.12**	**90.95**	**96.24**

As shown in Table 5.2, enlarging the replay buffer reduces forgetting for all methods, with this trend being especially marked on Sequential-CIFAR10. A similar effect is observed when increasing the temporal dimension from $T = 2$ to $T = 4$, which gen-

erally leads to more stable **SNN** behavior across configurations. Within this setting, **STAER** achieves the lowest forgetting in the large majority of cases. Notably, in the most constrained configurations, it is also able to outperform the corresponding **ANN** variants, indicating that, when combined with an effective replay strategy, the spiking formulation can offer strong robustness against forgetting.

Table 5.3 reports the results under the **TIL** setting. Also in this case, **STAER** outperforms the other spiking baselines in almost all configurations. Compared with **CIL**, the improvements obtained by increasing the temporal dimension are generally less pronounced, which is consistent with the fact that **TIL** is a less challenging scenario because task identity is available at inference time. Even so, **STAER** achieves performance that remains close to that of standard **ANN** methods. On Sequential-MNIST, the gap is limited across all buffer sizes, whereas on Sequential-CIFAR10 the difference is more noticeable, reflecting the higher complexity of the dataset.

5.4 Ablation

To better understand the contribution of the proposed *Temporal-Alignment* (**TA**) objective $\mathscr{L}_{TA}$, we perform an ablation study by varying its main hyperparameters. In particular, we sweep the alignment coefficient $\beta \in \{10^{-4}, 10^{-3}, 5 \times 10^{-3}\}$ and the *Soft-DTW divergence* contraction and expansion weights $\alpha_1, \alpha_2 \in \{0, 0.1, 0.25, 0.5, 0.75, 1.0\}$, while fixing $T = 2$. The analysis is conducted on Sequential-MNIST in the **CIL** setting, and results are reported in terms of final accuracy averaged over 5 random seeds.

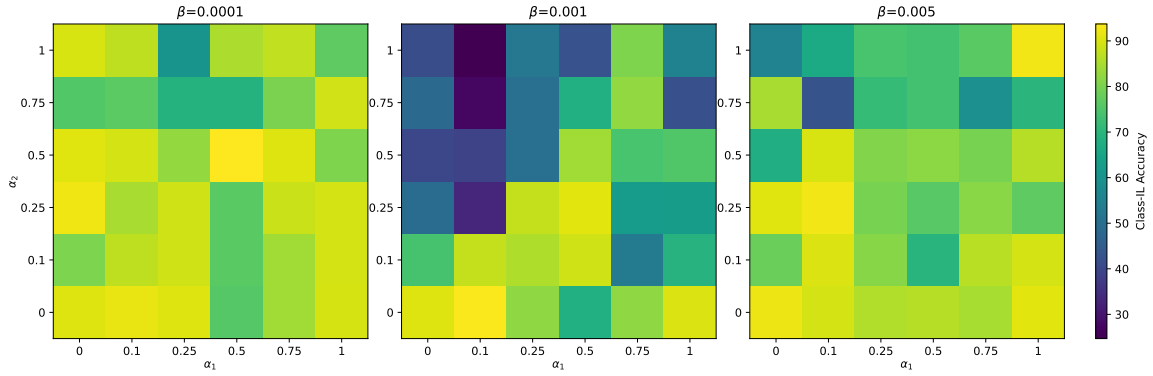

Figure 5.1: Hyperparameter sensitivity of the Temporal-Alignment objective on Sequential-MNIST. Each heatmap (fixed β) reports the final CIL accuracy (lighter color is better) as a function of the Soft-DTW Divergence compression/dilation weights (α_1, α_2) and the TA strength β. The x-axis shows α_1 (compression), the y-axis shows α_2 (dilation). Reproduced from [2].

As shown in Figure 5.1, the best performance is obtained when the alignment term is neither too weak nor too dominant, and when the contraction and expansion components are balanced. The strongest configuration reaches 93.71%, corresponding to $\beta = 10^{-4}$ and $\alpha_1 = \alpha_2 = 0.5$. For this value of β, the performance landscape appears relatively stable overall, although accuracy tends to deteriorate when the expansion term becomes too strong compared to the contraction one, namely for large values of α_2 not matched by α_1.

A different behavior emerges for $\beta = 10^{-3}$. In this case, the optimization becomes markedly more sensitive to the choice of α_1 and α_2. Although the best configuration still achieves a competitive result, namely 93.32% at $(\alpha_1, \alpha_2) = (0.1, 0)$, several combinations lead to severe degradation, with performance dropping as low as 24.69%. This trend further suggests that assigning excessive importance to dilation relative to compression can destabilize training.

Table 5.4: CIL accuracy on Sequential-MNIST enabling/disabling individual loss components to quantify their contribution. Reproduced from [2].

Setting	$\boldsymbol{\beta}$	$\boldsymbol{\alpha_1}$	$\boldsymbol{\alpha_2}$	**Class-IL**
snn-ER	✗	✗	✗	87.90
snn-ER + SDTW	✓	✗	✗	90.15
STAER w/o dilation	✓	✓	✗	91.02
STAER w/o compression	✓	✗	✓	91.69
STAER	✓	✓	✓	**93.71**

When the alignment strength is increased to $\beta = 5 \times 10^{-3}$, high peak performance is still attainable, up to 92.29% for $\alpha_1 = \alpha_2 = 1.0$. However, this regime remains less robust, since imbalanced settings continue to produce substantial drops in accuracy. Overall, these results indicate that the effectiveness of the **TA** objective depends not only on its overall weight, but also on maintaining a suitable balance between the two temporal transformations.

Finally, when setting $\alpha_1 = \alpha_2 = 0$, thus removing the contraction and expansion *Soft-DTW divergence* terms and retaining only the same-resolution alignment, the **FAA** decreases to 89.95% for $\beta = 10^{-4}$. This confirms that alignment across additional temporal resolutions provides a meaningful contribution to performance. The results reported in Table 5.4 further detail the role of each component of $\mathscr{L}_{TA}$. On the basis of this analysis, we selected $\beta = 10^{-4}$ and $\alpha_1 = \alpha_2 = 0.5$ for all the experiments reported in the previous section.

Chapter 6

Conclusion

6.1 Conclusion

This thesis investigated Continual Learning in **SNNs** by focusing on a property that is specific to spiking computation: past knowledge is not expressed only in the final prediction, but also in the temporal evolution of the response. Starting from this idea, we proposed **STAER**, a replay-based method that combines standard supervision with a *Temporal Alignment* objective based on *Soft-DTW divergence* and with multi-scale temporal traces.

The experimental results support this design. In the **CIL** setting, **STAER** consistently improves over the main replay-based spiking baselines, with the clearest gains appearing when the temporal dimension is increased to $T = 4$. In particular, on Sequential-CIFAR10 the method reaches 83.53% final accuracy, while on Sequential-MNIST it reaches 98.48%. Under **TIL**, the method remains strongly competitive and achieves

the best overall results in most configurations, up to 99.88% on Sequential-MNIST and 96.24% on Sequential-CIFAR10.

The forgetting analysis and the ablation study point in the same direction. Replay alone is useful, but the best results are obtained when past samples are preserved together with their temporal structure. In particular, the strongest configurations are those where contraction and expansion are balanced and where the alignment term remains moderate rather than dominant. This suggests that the main benefit of the method does not come only from storing previous examples, but from constraining how their responses evolve over time.

Overall, the thesis shows that the temporal dimension of **SNNs** can be exploited as a direct signal for Continual Learning. In this sense, the main contribution of the work is not only a practical improvement over existing spiking replay methods, but also the evidence that time-aware replay is a meaningful direction for reducing forgetting in incremental spiking systems.

6.2 Future Work

A natural extension of this work is to scale **STAER** beyond the current *ResNet19* backbone and study its behavior with stronger **CL** architectures. In particular, it would be interesting to combine the proposed *Temporal Alignment* mechanism with *transformer-based* continual learners.

A second direction concerns the evaluation protocol. In this thesis, the method is tested on standard sequential benchmarks, which are useful for controlled analysis but

only partially reflect the variability of real **CL** streams. For this reason, a meaningful next step would be to evaluate the method on more realistic settings such as Multi-domain Task Incremental Learning (**MTIL**), where the model must adapt not only to new classes or tasks but also to stronger domain shifts.

Finally, it would be worthwhile to explore the same principle on event-based neuromorphic datasets. We hope that the results and method showcased in this document offer some insights into Spiking Neural Networks for Continual Learning and that it will be useful for further research.

Bibliography

[1] G. M. van de Ven and A. S. Tolias, "Three scenarios for continual learning," *arXiv preprint arXiv:1904.07734*, 2019.

[2] M. Gianferrari, O. Moussadek, R. Salami, C. Fiorini, L. Tartarini, D. Gandolfi, and S. Calderara, "STAER: Temporal aligned rehearsal for continual spiking neural network," *arXiv preprint arXiv:2601.20870*, 2026.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[5] W. Gerstner and W. M. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[6] D. Garreau, R. Lajugie, S. Arlot, and F. Bach, "Metric learning for temporal sequence alignment," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 27, 2014, pp. 1817–1825.

[7] M. Cuturi and M. Blondel, "Soft-DTW: a differentiable loss function for time-series," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2017, pp. 894–903.

[8] M. Blondel, A. Mensch, and J.-P. Vert, "Differentiable divergences between time series," in *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2021, pp. 3853–3861.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[10] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[11] W. C. Abraham and A. Robins, "Memory retention—the synaptic stability versus plasticity dilemma," *Trends in Neurosciences*, vol. 28, no. 2, pp. 73–78, 2005.

[12] W. Maass, "Networks of spiking neurons: the third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[13] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks," *IEEE Signal Processing Magazine*, vol. 36, no. 6, pp. 51–63, 2019.

[14] A. L. Hodgkin and A. F. Huxley, "Currents carried by sodium and potassium ions through the membrane of the giant axon of Loligo," *The Journal of Physiology*, vol. 116, no. 4, pp. 449–472, 1952.

[15] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert, "iCaRL: Incremental classifier and representation learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2001–2010.

[16] R. Ratcliff, "Connectionist models of recognition memory: constraints imposed by learning and forgetting functions," *Psychological Review*, vol. 97, no. 2, pp. 285–308, 1990.

[17] A. Robins, "Catastrophic forgetting, rehearsal and pseudorehearsal," *Connection Science*, vol. 7, no. 2, pp. 123–146, 1995.

[18] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37–57, 1985.

[19] P. Buzzega, M. Boschini, A. Porrello, D. Abati, and S. Calderara, "Dark experience for general continual learning: a strong, simple baseline," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, pp. 15 920–15 930.

[20] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[21] H. Zheng, Y. Wu, L. Deng, Y. Hu, and G. Li, "Going deeper with directly-trained larger spiking neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, vol. 35, no. 12, 2021, pp. 11 062–11 070.

[22] S. Deng, Y. Li, S. Zhang, and S. Gu, "Temporal efficient training of spiking neural network via gradient re-weighting," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022. [Online]. Available: https://openreview.net/forum?id=_XNtisL32jv

[23] X. Yao, F. Li, Z. Mo, and J. Cheng, "GLIF: A unified gated leaky integrate-and-fire neuron for spiking neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022, pp. 32 160–32 171.

[24] K. Yu, C. Yu, T. Zhang, X. Zhao, S. Yang, H. Wang, Q. Zhang, and Q. Xu, "Temporal separation with entropy regularization for knowledge distillation in spiking neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025, pp. 8806–8816.

[25] W. Fang, Z. Yu, Y. Chen, T. Masquelier, T. Huang, and Y. Tian, "Incorporating learnable membrane time constant to enhance learning of spiking neural networks," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021, pp. 2641–2651.

[26] G. B. M. Mello, S. Soares, and J. J. Paton, "A scalable population code for time in the striatum," *Current Biology*, vol. 25, no. 9, pp. 1113–1122, 2015.

[27] H. Motanis and D. V. Buonomano, "Neural coding: time contraction and dilation in the striatum," *Current Biology*, vol. 25, no. 9, pp. R374–R376, 2015.

[28] F. Zenke, B. Poole, and S. Ganguli, "Continual learning through synaptic intelligence," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2017, pp. 3987–3995.

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015. [Online]. Available: https://arxiv.org/abs/1412.6980