Continual Learning with Gated Incremental Memories for sequential data processing

Andrea Cossu
Computer Science Dept.
University of Pisa
Pisa, Italy
andrea.cossu@sns.it

Antonio Carta
Computer Science Dept.
University of Pisa
Pisa, Italy
antonio.carta@di.unipi.it

Davide Bacciu
Computer Science Dept.
University of Pisa
Pisa, Italy
bacciu@di.unipi.it

***Abstract*—The ability to learn in dynamic, nonstationary environments without forgetting previous knowledge, also known as Continual Learning (CL), is a key enabler for scalable and trustworthy deployments of adaptive solutions. While the importance of continual learning is largely acknowledged in machine vision and reinforcement learning problems, this is mostly under-documented for sequence processing tasks. This work proposes a Recurrent Neural Network (RNN) model for CL that is able to deal with concept drift in input distribution without forgetting previously acquired knowledge. We also implement and test a popular CL approach, Elastic Weight Consolidation (EWC), on top of two different types of RNNs. Finally, we compare the performances of our enhanced architecture against EWC and RNNs on a set of standard CL benchmarks, adapted to the sequential data processing scenario. Results show the superior performance of our architecture and highlight the need for special solutions designed to address CL in RNNs.**

I. Introduction

Dynamic environments are often subjected to the concept drift phenomenon [1]–[5] which reflects substantial changes in the data generating process and the corresponding predictions. More formally, given an unknown time-dependent joint probability $p_t(\mathbf{y}, \mathbf{x})$ over data $\mathbf{x}$ and target $\mathbf{y}$, concept drift can affect both the evidence $p_t(\mathbf{x})$ and the conditional distribution $p_t(\mathbf{y}|\mathbf{x})$. Following the definitions presented in [5], this paper addresses the problem of learning in dynamic environments in which the evidence $p_t(\mathbf{x})$ exhibits instantaneous drifts without any time limit imposed on it, leading to a *permanent, abrupt concept drift*. Since one of the objectives of this paper is to evaluate the capability of a model to consolidate old knowledge without forgetting, we do not account for changes in the conditional distribution $p_t(\mathbf{y}|\mathbf{x})$ without corresponding changes in the evidence. In fact, this would lead to forgetting of the previous input-output mapping in favour of a new one. We will refer to a specific objective (e.g. learn to classify MNIST digits) as *task*. An *input distribution* $p_t(\mathbf{x})$, instead, will generate data related to a particular task (e.g. subsets of MNIST digits). Each task is associated to multiple input distributions (also called subtasks) which altogether define a dynamic environment in which the model is trained. We will deal with sequence classification tasks in which each input sequence $\mathbf{x} = [x_i]_{i=1,\dots,T}$, $x_i \in \mathbb{R}^d$ is associated to a scalar target $y \in \mathbb{R}$. Following the main trend in CL, the temporal boundary of each subtask is known to the model only at training time.

In the presence of dynamic environments with recurring concepts drifts [1] it is useful to design models that are able to recall and exploit previously acquired information. Unfortunately, continuous plasticity of internal representations under drifting task distributions is widely known to suffer from negative interference between the tasks that are incrementally presented to the model, yielding to the well known stability-plasticity dilemma [6] of connectionist models. The result is that the models catastrophically forget previously acquired knowledge [7] as new tasks become available.

The problem of catastrophic forgetting is the main focus of Continual Learning (CL), defined as *"the unending process of learning new things on top of what has already been learned"* [8].

In the CL scenario, a learning model is required to incrementally build and dynamically update internal representations as the distribution of tasks dynamically changes across its lifetime. Ideally, part of such internal representations will be general and invariant enough to be reusable across similar tasks, while another part should preserve and encode task-specific representations.

While current trends in CL put particular emphasis on computer vision applications or reinforcement learning scenarios, sequential data processing is rarely taken into consideration (see Section II). However, sequential data are heavily used in several fields like Natural Language Processing, signal processing, bioinformatics, and many others. In this context, Recurrent Neural Networks (RNNs) have the ability to develop neural representations that capture the history of their inputs. Learning proper memory representations is a major challenge for RNNs. In addition, in a CL setting, RNNs have to deal with drifts in task distributions which can greatly affect their capability of developing robust and effective memory representations.

We provide a threefold contribution to the discussion concerning CL in sequential data processing. First, we define a new dynamic approach, named Gated Incremental Memory (GIM), that imbues RNN architectures with CL skills by incre-

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

mentally adding new modules to capture the drifts in input distribution while avoiding catastrophic forgetting. GIM leverages autoencoders to automatically recognize input distributions and to select the correct module to process the sequence. Second, we apply Elastic Weight Consolidation (EWC) [9], a popular CL approach, on top of two different RNNs. At the best of our knowledge, we are the first to experiment with EWC on RNNs. Third, we test the performances of GIM against EWC and standard RNNs on three benchmarks originally introduced here by adapting traditional CL tasks to the sequential case. The results of our empirical analysis confirm the advantages of using our enhanced architecture over standard recurrent models. Such advantages are particularly clear when testing enhanced architecture on old distributions, since it successfully prevents forgetting. These results highlight some of the key differences between feedforward and recurrent CL techniques and pinpoint the need for solutions specifically designed for recurrent architectures.

II. Related Work

Learning in dynamic environments in the presence of concept drift has received much attention in the literature. Concept drift could be generated by hidden contexts [10], whose detection would result in drastic performance improvements. Predictions on incoming input patterns can be performed relying on a window of recently encountered instances (*instance selection*), thus accounting for drifts in the underlying distribution, like in FLORA systems [1]. Similarly, *instance weighting* systems [11] exploit weights on the input patterns which can inform predictions based on their similarity to specific concepts or simply their time obsolescence. Similar to the approach proposed in this paper, *ensemble methods* associate an expert to each (or to a group of) concepts and combine their predictions into the final answer [12, 13].

CL explores the problem of concept drift from a slightly different perspective, by focusing on how to avoid catastrophic forgetting on old distributions, while at the same time fostering learning of incoming data. The aspect of forgetting is peculiar of CL and it is at the center of our work.

CL literature mostly focuses on computer vision and reinforcement learning applications, with approaches ranging from regularization methods [9, 14], to dual models [15], to dynamic architectures [16, 17].

The first attempt to deal with sequential processing in CL was presented in [18], where the authors introduced a dual model rehearsed with pseudopatterns and trained to reconstruct the next element of a sequence (i.e. sequence modeling). More recently, RNNs have been exploited in combination with other techniques, such as Fixed Expansion Layer [19], external growing memories [20], Reservoir Computing [21] and backpropagation-free learning [22].

Evaluations of the performances of standard RNNs in CL are provided in [23] and [24]. However, while the latter provides no solutions to the problem of forgetting, the former introduces an effective, but rather complex, recurrent architecture, combining dynamic expansion approaches (*Net2Net*) with gradient projection on a reservoir of old samples (GEM). Instead, in addition to forgetting analysis, we propose a solution that mitigates its effects, while at the same time keeping our model simple enough to favor reproducibility as well as further extensions.

Our dynamic RNN architecture is inspired by Progressive networks [16], a popular CL approach used for feedforward networks that deals with drifts in the input distribution by dynamically expanding the existing model. In addition, we leverage gating autoencoders, introduced in [25] for feedforward architectures, to remove the need to know task identity at test time.

III. Gated Incremental Memories for continual learning with recurrent neural networks

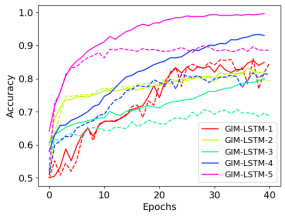
In this section, we introduce Gated Incremental Memory (GIM), a novel CL architecture designed for recurrent neural models and sequential data. In particular, we show how GIM can be obtained by combining a recurrent version of the Progressive network [16] and a set of gating autoencoders [25] to avoid, at test time, any explicit supervision about subtask labels. In the following, we denote an entire sequence with bold notation (e.g. $\mathbf{x}$), and a single vector with plain formatting (e.g. x_i).

A. Recurrent Neural Networks

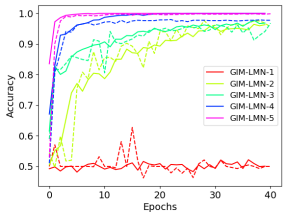
The proposed approach is independent of the underlying recurrent architecture. To highlight the generality of our approach, we focus our study on two different classes of RNNs, using either gated and non-gated approaches. Gated models, like LSTM [26], leverage adaptive gates to enable selective memory cell updates. In our analysis, we consider LSTM as a representative of gated architectures, given its popularity in literature and its state-of-the-art performances in several sequential data processing benchmarks. Non-gated approaches rely on different mechanisms to solve the vanishing gradient problem, like parameterizing recurrent connections with an orthogonal matrix [27]. In our analysis, we consider the Linear Memory Network (LMN) [28] as a representative of non-gated approaches. LMNs leverage a conceptual separation between a nonlinear feedforward mapping computing the hidden state h_t and a linear dynamic memory computing the history dependent state h_t^m. Briefly, in formulas:

$$h_t = \sigma(W^{xh}x_t + W^{mh}h_{t-1}^m)$$
$$h_t^m = W^{hm}h_t + W^{mm}h_{t-1}^m,$$

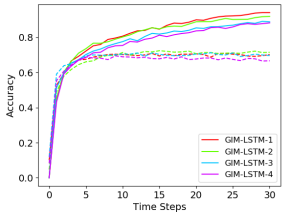
where x_t, h_t and h_t^m are the input vector, the functional component activation and the memory state at time t, respectively. The memory state h_t^m is the final output of the layer, which is given as input to subsequent layers. It is then useful to study how both models behave in CL environments and how their different memory representations respond to phenomena like drifting tasks distribution, eventually resulting



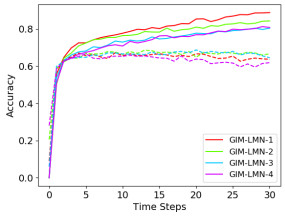
(a) GIM-LSTM on Devanagari: 40 epochs



(b) GIM-LMN on Devanagari: 40 epochs



(c) GIM-LSTM on Audioset: 30,000 time steps, printed every 1000 steps.



(d) GIM-LMN on Audioset: 30,000 time steps, printed every 1000 steps.

Fig. 1: Examples of accuracy curves on training set (solid line) and validation set (dashed line).

in catastrophic forgetting.

B. Elastic Weight Consolidation

Elastic Weight Consolidation (EWC) [9] is one of the most popular CL method. EWC mitigates forgetting by preventing large changes in those parameters which have been recognized important when learning previous distributions. Hence, in the case of recurring drifts, the model will still be able to address previous patterns without forgetting. In order to learn a new distribution, EWC builds on the assumption that, for an overparameterized model, it exists an optimal configuration of the parameters which is not too distant from the current one in the parameters space and which is able to adapt to upcoming drifts.

The importance of each connection is estimated through an approximation of the diagonal of the Fisher Information Matrix, whose computation requires only first-order derivatives. The penalization is implemented by adding a quadratic regularization term to the standard loss function, weighted by the previously computed connection importance. Given task A already learned by the model, the loss function when learning a new task B is:

$$L(\theta) = L_B(\theta) + \sum_i \frac{\lambda}{2} F_i(\theta_i - \theta^*_{A,i})^2, \tag{1}$$

where F_i is the i-th diagonal element of the (approximated) Fisher Information Matrix, L_B is the loss for task B, θ are the model parameters, θ^*_A are the learned parameters for task A, λ is the hyperparameter controlling the tradeoff between accuracy on old and new task.

We implemented EWC in RNNs and we observed its performances in terms of forgetting and accuracy on our benchmarks. At the best of our knowledge, we are the first to study EWC in a sequential context with RNNs.

C. Gated Incremental Memory (GIM)

GIM is a general class of dynamic, recurrent architectures that can be built on top of any recurrent model. GIM relies on a progressive memory [16] extension of the underlying RNN model, which uses separate modules for each subtask. It also leverages a set of gating autoencoders, one for each subtask, to automatically select the module that best matches the current input. Figures 2 and 3 provide an overview of the entire GIM architecture during training and test.

1) RNN Modules: The main component of GIM is the RNN *module*. As soon as a new distribution arrives, a new RNN module is added on top of the existing architecture and connected to the previous one (Fig. 2). The exact inter-modules connections are slightly different depending on the underlying recurrent model. When using the GIM-LSTM, at each timestep, the new module takes as additional input the current hidden state of the previous module. Instead, the GIM-LMN takes as additional input the concatenation of the previous module's memory h^m_t and functional activation h_t. These additional inputs allow to easily transfer knowledge from the previous modules to the new ones. To prevent forgetting, when a new module is added to the existing architecture, the previous module's parameters are frozen and no longer updated. Therefore, each module becomes an expert of its own domain. At each timestep t, the input vector x_t is forwarded to all modules. Each module has its own output layer and, during training, the last module added to the network is used to generate the final output y from the last hidden state of the module. Given an input $\mathbf{x} = x_1, \ldots, x_T$, the output y for a GIM-LMN with N modules can be computed as follows:

$$\begin{aligned} h^m_{:,1}, h_{:,1} &= \text{LMN}_1(\mathbf{x}, h^m_{0,1}) \\ h^m_{:,j}, h_{:,j} &= \text{LMN}_j([\mathbf{x}; h^m_{:,j-1}; h_{:,j-1}], h^m_{0,j}), \quad j = 2, ..., N \\ y &= \sigma(W^{mo}_j h^m_{T,N}), \end{aligned}$$

where LMN_j is the RNN module corresponding to the j-th subtask, $h^m_{:,j}$ and $h_{:,j}$ are the sequences of memory states and functional activations of module j (: indexes all the steps in the sequence), and $[\cdot;\cdot]$ is the concatenation operator between vectors. The aggregated output y is computed by passing the final memory state $h^m_{T,N}$ through a linear layer.

LSTM modules follow the same logic for the forward pass, substituting the final hidden state h_T to the memory state and functional activations of GIM-LMN. A detailed description is provided in Algorithm 1 for LSTM modules and Algorithm 2 for LMN modules.

2) Gating Autoencoders: At inference time, GIM models must choose which module to use to compute the output. To solve this problem, each module is associated with an LSTM

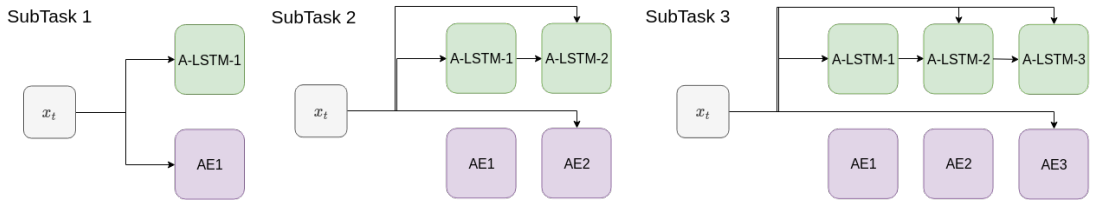


Fig. 2: Incremental expansion of the GIM-LSTM during training on 3 subtasks. When a new subtask is encountered a new module is added.

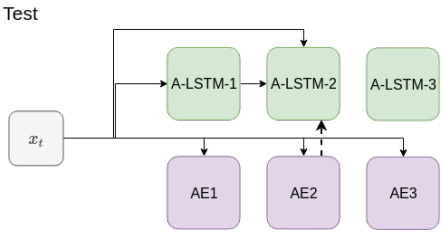


Fig. 3: GIM-LSTM at inference time. The input $\mathbf{x}$ is encoded by all the autoencoders. The autoencoder with the minimum reconstruction error (AE2 in the example) determines which module to choose (LSTM-2 in the example). The input is passed to the chosen module to compute the output (dashed line).

autoencoder (AE) [29], which is a sequence-to-sequence model trained to encode and reconstruct the input sequence $\mathbf{x}$. Each autoencoder is trained only on data from the subtask used for the corresponding module. Algorithm 3 shows the procedure to reconstruct the input using the AE encoder and decoder.

3) Training: GIM is trained sequentially on each subtask, by adding and training a new module whenever a new task is encountered. The current RNN module is trained by minimizing the Cross Entropy loss for classification tasks, while the corresponding autoencoder is trained to minimize the reconstruction error, by optimizing the mean squared error (MSE) between the input and the reconstructed sequence. The previous modules and autoencoders are not trained anymore and their parameters remain constant. Algorithm 3 shows the pseudocode for the training procedure.

4) Inference: At inference time, the computation proceeds in three steps:

1) the autoencoders reconstruct the input sequence;
2) the subtask is identified by selecting the autoencoder with the minimum reconstruction error;
3) the module corresponding to the identified subtask is used to compute the output.

The inference procedure is detailed by the following equations:

$$\tilde{\mathbf{x}}_i = \text{AE}_i(\mathbf{x}), i = 1, ..., N$$
$$k = \arg\min_i \text{MSE}(\tilde{\mathbf{x}}_i, \mathbf{x})$$
$$y = \text{LMN}_k(\mathbf{x}).$$

The algorithms describing the output computation at inference time are in Algorithm 3.

Algorithm 1 GIM-LSTM Forward Pass for Module N

function LSTM-MODULE-FW(GIM, $\mathbf{x}$, N)
Require: GIM-LSTM with at least N modules, $N \geq 1$, $\mathbf{x}$ with T timesteps
  $h_{0,1} \leftarrow 0$
  $h_{:,1} \leftarrow \text{GIM.LSTM}_1(\mathbf{x}, h_{0,1})$
  for $d \leftarrow 2, N$ **do**
    $h_{0,d} \leftarrow 0$
    $\hat{\mathbf{x}} \leftarrow [\mathbf{x}; h_{:,d-1}]$
    $h_{:,d} \leftarrow \text{GIM.LSTM}_d(\hat{\mathbf{x}}, h_{0,d})$
  end for
  $y \leftarrow W^{out} h_{T,N}$
  return y
end function

Algorithm 2 GIM-LMN Forward Pass for Module N

function LMN-MODULE-FW(GIM, $\mathbf{x}$, N)
Require: GIM-LMN with at least N modules, $N \geq 1$, $\mathbf{x}$ with T timesteps
  $h^m_{0,1} \leftarrow 0$
  $h^m_{:,1}, h_{:,1} \leftarrow \text{GIM.LMN}_1(\mathbf{x}, h^m_{0,1})$
  for $d \leftarrow 2, N$ **do**
    $h^m_{0,d} \leftarrow 0$
    $\hat{\mathbf{x}} \leftarrow [\mathbf{x}; h^m_{:,d-1}; h_{:,d-1}]$
    $h^m_{:,d}, h_{:,d} \leftarrow \text{GIM.LMN}_d(\hat{\mathbf{x}}, h^m_{0,d})$
  end for
  $y^m_N \leftarrow W^{mo}_N h^m_{T,N}$
  return y^m_N
end function

IV. Advantages of the GIM architecture

GIM, like Progressive networks, is capable of learning multiple distributions without being affected by forgetting. Freezing old parameters easily guarantees that the model will retain the knowledge about previous subtasks, while the use of the activations of the previous module as additional inputs allow to transfer knowledge from the previous model to the new ones. Additionally, GIM overcomes one of the major drawbacks of Progressive networks [16]: it does not require

Algorithm 3 Functions to compute the reconstruction of the autoencoder, for training it, and for choosing the GIM module.

function RECONSTRUCTION(AE, $\mathbf{x}$)
  $h_{enc,0} \leftarrow 0$
  $h_{enc,:} \leftarrow \text{AE.LSTM}_{enc}(\mathbf{x}, h_{enc,0})$
  $h_{dec,:} \leftarrow \text{AE.LSTM}_{dec}(\mathbf{0}, h_{enc,T})$
  $\tilde{\mathbf{x}} \leftarrow \text{AE.}W^{out} h_{dec,:}$
  return $\tilde{\mathbf{x}}$
end function
function AE-TRAIN($\mathcal{D}$)
Require: $\mid \mathcal{D} \mid > 1$
  $l_{ae} \leftarrow []$
  while a new distribution D_k is available **do**
    AE $\leftarrow$ init-autoencoder()
    l_{ae}.append(AE)
    for training batch $\mathbf{x} \in D_k$ **do**
      $\tilde{\mathbf{x}} \leftarrow$ RECONSTRUCTION(AE, $\mathbf{x}$)
      $J \leftarrow \text{MSE}(\mathbf{x}, \tilde{\mathbf{x}})$
      $\frac{\partial J}{\partial w} \leftarrow$ backprop(J)
      Take a descent step along $\frac{\partial J}{\partial w}$
    end for
  end while
  return l_{ae}
end function
function GIM-INFERENCE(GIM, $\mathbf{x}$)
  $l_{rec} \leftarrow []$
  for AE $\in$ GIM.l_{ae} **do**
    $\tilde{\mathbf{x}} \leftarrow$ RECONSTRUCTION(AE, $\mathbf{x}$)
    $l_{rec}.append(\text{MSE}(\mathbf{x}, \tilde{\mathbf{x}}))$
  end for
  $m \leftarrow \arg\min l_{rec}$ ▷ index of the best autoencoder
  $y \leftarrow$ LSTM-MODULE-FW(GIM, $\mathbf{x}, m$)
  return y
end function

explicit knowledge about input distributions at test time, since gating autoencoders are able to autonomously recognize the current input and use the appropriate module to compute the output. Compared to Progressive networks, GIM simplifies the inter-modules connections: while Progressive networks use feedforward networks, created between a module and *all* the next ones, GIM employs only concatenation between vectors and connects only adjacent modules. Since both Progressive and GIM employ a separate module for each of the n subtasks, the number of adaptive parameters in the architecture is quadratic in n for the Progressive network, while it is linear for GIM. Given $\Theta_M = \{\theta_i | \theta_i \in M\}$, where M is a generic model, we obtain the following upper bounds:

$$|\Theta_{Progressive}| = \mathcal{O}(n^2), \quad |\Theta_{GIM}| = \mathcal{O}(n).$$

V. DATASETS

We experimented with three different datasets: following the current trend in CL, two of them, MNIST and Devanagari, originate from images. The third one, Audioset, is constructed by processing short clips of audio sounds and is therefore more representative of a sequential, dynamic environment[1].
MNIST and Devanagari are adapted to sequential data processing by transforming each image in a sequence of pixels, which are then shuffled according to a fixed, random permutation. Permuting the images ensures that the RNN performance is not affected by the long sequence of non-informative elements (pixels) which are present at the end of each sequence. Considering the image resolution of 28x28, the sequences consist of 784 timesteps, making these datasets challenging not only for the CL scenario but also for recurrent models in general due to the length of the input sequences.
In order to create dynamic environments, we choose to follow the standard approach of CL literature [23, 24] by dividing each dataset into groups of non-overlapping classes, which we call subtasks. When training on a specific dataset (i.e. when addressing a particular task), subtasks are presented to the model sequentially one after the other (the next one starts when the previous has ended).
Concept drift is present on the output layer of each model since the input distribution associated with each output unit changes from one subtask to the next. The model should adapt to the drift and learn the new concept without forgetting the previous ones. When finished training on all subtasks, the model is tested on both last and previous subtasks to assess its resilience to forgetting.

A. MNIST

One of the most used datasets in Machine Learning and CL is the MNIST dataset of handwritten digit [30]. Each image has size 28x28, gray scaled, which leads to input sequences with $28 \cdot 28 = 784$ scalars. We created 5 subtasks, corresponding to the 5 digits partitions: $(0, 1), (2, 3), (4, 5), (6, 7), (8, 9)$ and we trained the models to classify each pair of digits, switching to the next subtasks once the previous one is completed.

B. Devanagari

Devanagari is a dataset composed of images of handwritten characters belonging to 46 different classes [31]. Each class has $1,700$ images, each of which of size 32x32, gray-scaled. Following the approach of [24], we randomly selected 10 classes out of the 46. In addition, we also remove the padding along the borders of the image, resulting in 28x28 gray-scaled input images. The total length of the input sequence is therefore the same as in MNIST.
We split the selected classes in 5 subtasks of 2 classes each: (*gha*, *cha*), (*chha*, *daa*), (*bha*, *ma*), (*motosaw*, *petchiryakha*), (*1*, *3*). The last subtask is composed of digits, which however have a completely different representation than the ones in MNIST. The objective of the task is to assign to each sequence the correct class.

[1]Code to reproduce results is available at https://github.com/AndreaCossu/ContinualLearning-SequentialProcessing

TABLE I: Validation (top of cell) and Test (bottom of cell) accuracy ($\pm$ std) on all datasets (D) and subtasks (S). Validation accuracy on each subtask computed after training on that specific subtask. Test accuracy computed at the end of training on all subtasks. For each dataset, final row **Mean** shows Validation / Test accuracy averaged over all subtasks. Results averaged over 5 runs.

D	S	LSTM	LMN	EWC-LSTM	EWC-LMN	GIM-LSTM	GIM-LMN
MNIST	1	0.97 ± 0.01 0.55 ± 0.03	0.99 ± 0.02 0.45 ± 0.05	0.95 ± 0.01 0.74 ± 0.06	0.98 ± 0.02 0.32 ± 0.08	0.97 ± 0.041 0.97 ± 0.09	0.99 ± 0.01 $\mathbf{0.98 \pm 0.07}$
	2	0.86 ± 0.03 0.47 ± 0.07	0.97 ± 0.02 0.58 ± 0.06	0.72 ± 0.04 0.54 ± 0.08	0.88 ± 0.04 0.63 ± 0.07	0.92 ± 0.04 0.50 ± 0.08	0.98 ± 0.02 $\mathbf{0.98 \pm 0.09}$
	3	0.94 ± 0.04 0.18 ± 0.08	0.99 ± 0.05 0.14 ± 0.04	0.62 ± 0.03 0.43 ± 0.03	0.80 ± 0.03 $\mathbf{0.45 \pm 0.08}$	0.93 ± 0.01 0.35 ± 0.08	0.99 ± 0.02 0.35 ± 0.09
	4	0.98 ± 0.03 0.75 ± 0.06	0.99 ± 0.04 0.76 ± 0.04	0.56 ± 0.03 0.54 ± 0.09	0.93 ± 0.06 0.76 ± 0.08	0.96 ± 0.05 0.91 ± 0.12	0.99 ± 0.02 $\mathbf{0.96 \pm 0.09}$
	5	0.94 ± 0.01 0.94 ± 0.06	0.98 ± 0.03 0.98 ± 0.03	0.65 ± 0.05 0.64 ± 0.05	0.71 ± 0.04 0.72 ± 0.06	0.88 ± 0.01 0.76 ± 0.08	0.97 ± 0.04 $\mathbf{0.95 \pm 0.07}$
	Mean (V/T)	0.94 / 0.58	0.98 / 0.58	0.70 / 0.58	0.86 / 0.58	0.93 / 0.70	0.98 / **0.84**
Devanagari	1	0.82 ± 0.04 0.48 ± 0.03	0.50 ± 0.04 0.55 ± 0.06	0.80 ± 0.05 $\mathbf{0.60 \pm 0.03}$	0.50 ± 0.05 0.52 ± 0.04	0.82 ± 0.01 0.59 ± 0.02	0.50 ± 0.07 0.39 ± 0.06
	2	0.76 ± 0.07 0.44 ± 0.05	0.85 ± 0.07 0.59 ± 0.03	0.76 ± 0.06 0.48 ± 0.07	0.50 ± 0.08 0.49 ± 0.04	0.81 ± 0.11 $\mathbf{0.74 \pm 0.07}$	0.96 ± 0.08 0.73 ± 0.07
	3	0.65 ± 0.09 0.49 ± 0.08	0.87 ± 0.03 0.69 ± 0.03	0.63 ± 0.05 0.55 ± 0.05	0.49 ± 0.08 0.56 ± 0.08	0.71 ± 0.07 0.67 ± 0.09	0.95 ± 0.05 $\mathbf{0.86 \pm 0.10}$
	4	0.76 ± 0.05 0.49 ± 0.04	0.98 ± 0.03 0.35 ± 0.05	0.71 ± 0.07 $\mathbf{0.60 \pm 0.06}$	0.50 ± 0.06 0.48 ± 0.09	0.79 ± 0.08 0.51 ± 0.08	0.98 ± 0.04 0.33 ± 0.08
	5	0.90 ± 0.05 0.91 ± 0.05	0.99 ± 0.04 0.99 ± 0.06	0.86 ± 0.07 0.82 ± 0.06	0.51 ± 0.08 0.52 ± 0.4	0.87 ± 0.03 0.66 ± 0.09	0.99 ± 0.05 $\mathbf{0.83 \pm 0.03}$
	Mean (V/T)	0.78 / 0.56	0.84 / 0.63	0.75 / 0.61	0.50 / 0.51	0.80 / **0.63**	0.88 / **0.63**
Audioset	1	0.63 ± 0.01 0.10 ± 0.02	0.60 ± 0.01 0.05 ± 0.02	0.67 ± 0.03 0.08 ± 0.04	0.61 ± 0.03 0.04 ± 0.01	0.68 ± 0.02 $\mathbf{0.64 \pm 0.04}$	0.62 ± 0.03 0.55 ± 0.01
	2	0.71 ± 0.04 0.09 ± 0.02	0.67 ± 0.03 0.08 ± 0.01	0.71 ± 0.03 0.12 ± 0.02	0.65 ± 0.03 0.14 ± 0.02	0.73 ± 0.03 $\mathbf{0.71 \pm 0.03}$	0.68 ± 0.02 0.65 ± 0.08
	3	0.68 ± 0.03 0.13 ± 0.03	0.64 ± 0.01 0.14 ± 0.2	0.68 ± 0.01 0.13 ± 0.02	0.64 ± 0.01 0.15 ± 0.01	0.71 ± 0.04 $\mathbf{0.57 \pm 0.03}$	0.63 ± 0.03 0.54 ± 0.02
	4	0.67 ± 0.01 0.46 ± 0.02	0.63 ± 0.02 0.43 ± 0.02	0.62 ± 0.03 0.50 ± 0.01	0.62 ± 0.02 0.47 ± 0.02	0.67 ± 0.03 $\mathbf{0.50 \pm 0.03}$	0.63 ± 0.04 0.42 ± 0.05
	Mean (V/T)	0.67 / 0.20	0.64 / 0.18	0.67 / 0.21	0.63 / 0.20	0.70 / **0.61**	0.64 / 0.54

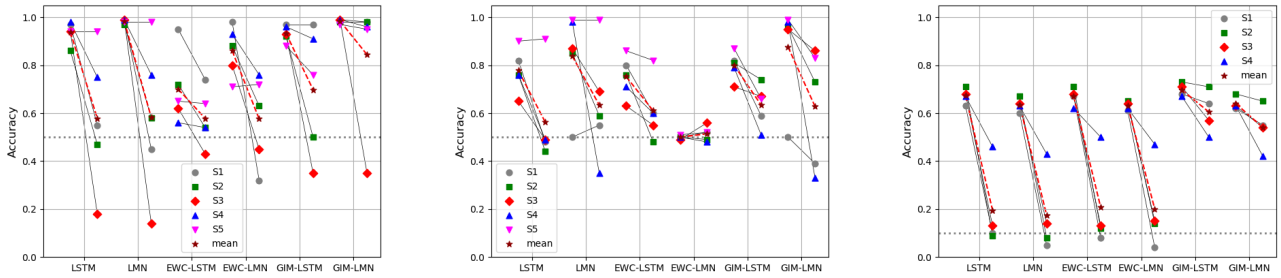
C. Audioset

Audioset [32] is a collection of annotated audio events, extracted from 10 seconds audio clips and organized hierarchically in classes. The objective is the classification of a sound from its audio clip source, embedded through a VGG-acoustic model into 10 vectors, one per second, each of dimension 128. To implement a CL scenario, we selected 40 audio classes and split them among 4 subtasks (10 classes per subtask). We selected the 40 classes according to the procedure outlined by [33]. Since the authors did not publish the classes, we randomly selected them from the superset resulting from their preprocessing pipeline. Audioset data has already been used in literature to assess CL skills [33]. However, the authors focused on the task from a *static* perspective, relying on the use of feedforward models only. Since the preprocessing step provides, for each audio clip, a sequence of fixed-size embeddings, it is possible to concatenate the vectors into a single large vector and feed it to the network. The sequential aspect of the task, however, is completely lost. At the best of our knowledge, we are the first to tackle Audioset in CL scenarios with recurrent models. It is also important to notice that the task difficulty is increased when using recurrent networks, since the model is not able to see the input in its entirety (like in feedforward networks), but it has to scan it one timestep at a time.

VI. Experiments

On MNIST and Devanagari we trained all models with Adam optimizer [34], learning rate of $1e-4$, mini batch size of 32. The number of hidden units is set to 128 for both LSTM-based and LMN-based (functional and memory component) models.
It is well known that orthogonal initialization of memory weight matrixes can improve learning when dealing with long sequences and linear memories [27, 35]. Therefore, we chose to use such initialization for LMN-based models and also to preserve it during training through an additional penalty in the loss function, expressed by $\beta\|(W^{mm})^T W^{mm} - I\|^2$, where W^{mm} is the memory weight matrix of the LMN, I is the identity matrix and β is the hyperparameter associated to the penalization. We used a value of $\beta = 0.1$, which was capable of preserving orthogonality on all experiments.

On Audioset we adopted a different configuration by using smaller models with 16 hidden units on LSTM and LMN (memory and functional component). This choice was determined by the limited amount of data points available for each class in Audioset: larger models led to overfitting without any



(a) MNIST (b) Devanagari (c) Audioset

Fig. 4: Paired plots for the three tasks. Each pair plot shows, for each model and for each subtask, mean validation accuracy (left point) computed after training on that subtask, and mean test accuracy (right point) computed at the end of the entire training on all subtasks. Validation and test accuracies are connected by a line. Therefore, the drop in performance due to forgetting is the difference between the two points. The red dashed line is the average among subtasks. Horizontal dotted line is equal to random classifier performance (0.5 for MNIST and Devanagari, 0.1 for Audioset).

performance improvements. We used the RMSProp optimizer, with learning rate of $3e-5$, momentum of 0.9 and L2 regularization with hyperparameter of $1e-3$ and mini batch size of 4.

LSTM autoencoders use 500 hidden units on encoder and decoder (36.22% compression on MNIST and Devanagari, 60.94% on Audioset) trained with Adam optimizer and learning rate of $1e-4$. On Audioset, we also use L2 regularization with hyperparameter $1e-3$.

We compare GIM-LSTM and GIM-LMN with standard LSTM and LMN and with the popular EWC method [9]. EWC requires to choose the value of the hyperparameter regulating the tradeoff between new incoming subtask and older ones. We choose the value of 0.4 out of $0.01, 0.1, 0.4, 1.0$, since it gave the best performances on a held-out validation set.

Table I provides the results of our experiments, averaged over 5 runs. Paired plots (Fig. 4) show the comparison between the validation performance for each subtask computed after training on that subtask (on the left) and the test performance computed after the final training is completed for all the subtasks. Therefore, the forgetting for each configuration can be evaluated by looking at the difference between the left and right points for each subtask.

MNIST and Devanagari are binary classification tasks, hence a random classifier would score 0.50 accuracy on all subtasks. On Audioset, being composed of 10 classes per subtask, a random classifier would score 0.10 accuracy on all subtasks.

We also show examples of learning curves, comparing GIM-LMN with GIM-LSTM (Fig. 1). Audioset shows early overfitting even with small models. A similar behavior, even if less drastic, is detected on Devanagari. It is, however, important to stress the fact that the learning curves cannot show the effect of forgetting because they are computed using the data from the current subtask, while we are interested in the final test accuracy, measured after training on all subtasks.

VII. Discussion

Table I and Figure 4 show that, in accordance with the results presented in [24], LSTM and LMN models suffer from catastrophic forgetting of old subtasks, independently of the performance achieved on the validation set during training. Even models regularized with EWC are not able to mitigate catastrophic forgetting. Notably, EWC always lowers the performance of the last subtask, an effect that is probably caused by the strong regularization imposed on the model weights. We were unable to find an EWC setting capable of guaranteeing a good tradeoff between current and previous subtasks accuracies. We hypothesize that the recurrence in RNNs could be the cause of the poor performance of EWC, leading to an importance evaluation through the Fisher Information matrix which is not representative of the (sub)task on which the model is trained. However, further studies will be needed to validate or contradict this hypothesis.

GIM models are by far the best performing ones, since they successfully learn on dynamic environments while limiting forgetting.

On Audioset, GIM-LSTM and GIM-LMN are capable of maintaining comparable performance on all subtasks once training is finished, while performance for standard and EWC-based models drops below the random baseline for some subtasks. This means that the autoencoders successfully recognize the incoming distribution and select the correct module to produce the output without any information on the incoming input labels.

On MNIST we obtained similar results, with some exceptions: GIM-LSTM underwent complete forgetting on 2 subtasks out of 5, while GIM-LMN experienced it on 1 out of 5. In those cases, autoencoders failed to reconstruct the input sequences, leading to the choice of the wrong module for the final classification. On these subtasks, the EWC version of LMN and LSTM surpassed GIM architectures.

Devanagari is the most challenging task: GIM models still

exhibited complete forgetting on 2 subtasks out of 5, with reduced performances on almost all the others. However, this behavior is common to all models on this task. EWC-LMN does not show significant differences between performances on the validation set and final test only because it is unable to learn the task, achieving an accuracy equivalent to the one of a random classifier.

VIII. Conclusions

The main objective of this work is to draw attention to the problem of CL for sequential data processing by introducing GIM, a recurrent CL architecture inspired by the Progressive networks [16]. Our benchmarks show that GIM is able to mitigate forgetting on computer vision and sequential audio data. GIM surpasses LSTM, LMN and their corresponding EWC version on the large majority of the experiments. The performance of GIM models depends on the reconstruction error of the subtask's autoencoders. In the future, different models for the autoencoders could be used to further improve the performance. The comparison with EWC on sequential, dynamic environments supports the claim that recurrent architectures need to be adapted to manage CL scenarios and encourages future works towards an in-depth study of their behaviors.

References

[1] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," *Machine Learning*, vol. 23, pp. 69–101, Apr. 1996.

[2] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Computing Surveys (CSUR)*, vol. 46, pp. 44:1–44:37, Mar. 2014.

[3] A. Tsymbal, "The Problem of Concept Drift: Definitions and Related Work," tech. rep., Trinity College, Dublin, 2004.

[4] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under Concept Drift: A Review," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, pp. 2346–2363, Dec. 2019.

[5] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar, "Learning in Non-stationary Environments: A Survey," *IEEE Computational Intelligence Magazine*, vol. 10, pp. 12–25, Nov. 2015.

[6] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, vol. 113, pp. 54–71, May 2019.

[7] R. French, "Catastrophic forgetting in connectionist networks," *Trends in Cognitive Sciences*, vol. 3, pp. 128–135, Apr. 1999.

[8] M. Ring, "Recurrent Transition Hierarchies for Continual Learning: A General Overview," *Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[9] 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," *PNAS*, vol. 114, no. 13, pp. 3521–3526, 2017. arXiv: 1612.00796.

[10] M. B. Harries, C. Sammut, and K. Horn, "Extracting Hidden Context," *Machine Learning*, vol. 32, pp. 101–126, Aug. 1998.

[11] R. Klinkenberg, "Learning drifting concepts: Example selection vs. example weighting," *Intelligent Data Analysis*, vol. 8, pp. 281–300, Aug. 2004.

[12] J. C. Schlimmer and R. H. Granger, "Incremental learning from noisy data," *Machine Learning*, vol. 1, pp. 317–354, Sept. 1986.

[13] K. O. Stanley, *Learning Concept Drift with a Committee of Decision Trees*. 2001.

[14] F. Zenke, B. Poole, and S. Ganguli, "Continual Learning Through Synaptic Intelligence," in *International Conference on Machine Learning*, pp. 3987–3995, July 2017.

[15] G. E. Hinton and D. C. Plaut, "Using Fast Weights to Deblur Old Memories," *Proceedings of the ninth annual conference of the Cognitive Science Society*, pp. 177–186, 1987.

[16] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive Neural Networks," *arXiv: 1606.04671 [cs]*, June 2016. arXiv: 1606.04671.

[17] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong Learning With Dynamically Expandable Networks," *ICLR*, p. 11, 2018.

[18] B. Ans, S. Rousset, R. M. French, and S. C. Musca, "A dual-network architecture with self-refreshing memory to overcome catastrophic forgetting in multiple sequence learning," 2002.

[19] R. Coop and I. Arel, "Mitigation of catastrophic forgetting in recurrent neural networks using a Fixed Expansion Layer," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, (Dallas, TX, USA), pp. 1–7, IEEE, Aug. 2013.

[20] N. Asghar, L. Mou, K. A. Selby, K. D. Pantasdo, P. Poupart, and X. Jiang, "Progressive Memory Banks for Incremental Domain Adaptation," *arXiv: 1811.00239 [cs]*, Nov. 2018. arXiv: 1811.00239.

[21] T. Kobayashi and T. Sugino, "Continual Learning Exploiting Structure of Fractal Reservoir Computing," in *Artificial Neural Networks and Machine Learning – ICANN 2019: Workshop and Special Sessions* (I. V. Tetko, V. Kůrková, P. Karpov, and F. Theis, eds.), vol. 11731, pp. 35–47, Cham: Springer International Publishing, 2019.

[22] A. Ororbia, A. Mali, C. L. Giles, and D. Kifer, "Continual Learning of Recurrent Neural Networks by Locally Aligning Distributed Representations," *arXiv:1810.07411 [cs]*, Aug. 2019. arXiv: 1810.07411.

[23] S. Sodhani, S. Chandar, and Y. Bengio, "On Training Recurrent Neural Networks for Lifelong Learning," *arXiv: 1811.07017 [cs, stat]*, Nov. 2018. arXiv: 1811.07017.

[24] M. Schak and A. Gepperth, "A Study on Catastrophic Forgetting in Deep LSTM Networks," in *Artificial Neural Networks and Machine Learning – ICANN 2019: Deep Learning* (I. V. Tetko, V. Kůrková, P. Karpov, and F. Theis, eds.), Lecture Notes in Computer Science, (Cham), pp. 714–728, Springer International Publishing, 2019.

[25] R. Aljundi, P. Chakravarty, and T. Tuytelaars, "Expert Gate: Lifelong Learning with a Network of Experts," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7120–7129, July 2017.

[26] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[27] Z. Mhammedi, A. Hellicar, A. Rahman, and J. Bailey, "Efficient Orthogonal Parametrisation of Recurrent Neural Networks Using Householder Reflections," in *International Conference on Machine Learning*, pp. 2401–2409, July 2017.

[28] D. Bacciu, A. Carta, and A. Sperduti, "Linear Memory Networks," in *Proceedings of the 28th International Conference on Artificial Neural Networks (ICANN 2019),*, Lecture Notes in Computer Science, Springer-Verlag, Sept. 2019.

[29] N. Srivastava, E. Mansimov, and R. Salakhutdinov, "Unsupervised Learning of Video Representations using LSTMs," *ICML*, 2015. arXiv: 1502.04681.

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[31] S. Acharya, A. K. Pant, and P. K. Gyawali, "Deep Learning Based Large Scale Handwritten Devanagari Character Recognition," *Proceedings of the 9th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*, pp. 121–126, 2015.

[32] J. F. Gemmeke, D. P. W. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal, and M. Ritter, "Audio Set: An ontology and human-labeled dataset for audio events," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 776–780, Mar. 2017.

[33] R. Kemker, M. McClure, A. Abitino, T. L. Hayes, and C. Kanan, "Measuring Catastrophic Forgetting in Neural Networks," in *Thirty-Second AAAI Conference on Artificial Intelligence*, Apr. 2018.

[34] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2015.

[35] M. Henaff, A. Szlam, and Y. LeCun, "Recurrent Orthogonal Networks and Long-Memory Tasks," in *International Conference on Machine Learning*, pp. 2034–2042, June 2016.