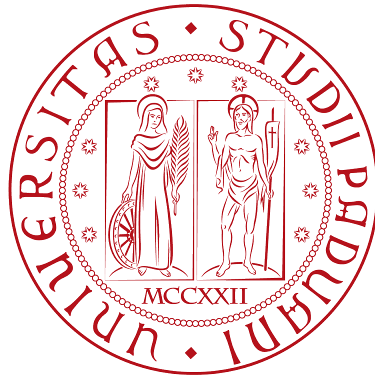
Università degli Studi di Padova

Dipartimento di Matematica
"Tullio Levi-Civita"

Laurea Magistrale in Informatica



Extending Gated Linear Networks for Continual Learning

Tesi di Laurea Magistrale

Relatore
Prof. Nicolò Navarin

Laureando
Matteo Munari

Anno Accademico 2021-2022

Matteo Munari: *Extending Gated Linear Networks for Continual Learning,* Tesi di Laurea Magistrale, © Aprile 2022.

“If a machine is expected to be infallible, it cannot also be intelligent.”

— Alan Turing

A Giulia

Abstract

To incrementally learn multiple tasks from an indefinitely long stream of data is a real challenge for traditional machine learning models. If not carefully controlled, the learning of new knowledge strongly impacts on a model's learned abilities, making it to forget how to solve past tasks.

Continual learning faces this problem, called catastrophic forgetting, developing models able to continually learn new tasks and adapt to changes in the data distribution.

In this dissertation, we consider the recently proposed family of continual learning models, called Gated Linear Networks (GLNs), and study two crucial aspects impacting on the amount of catastrophic forgetting affecting gated linear networks, namely, data standardization and gating mechanism.

Data standardization is particularly challenging in the online/continual learning setting because data from future tasks is not available beforehand. The results obtained using an online standardization method show a considerably higher amount of forgetting compared to an offline –static– standardization. Interestingly, with the latter standardization, we observe that GLNs show almost no forgetting on the considered benchmark datasets.

Secondly, for an effective GLNs, it is essential to tailor the hyperparameters of the gating mechanism to the data distribution. In this dissertation, we propose a gating strategy based on a set of prototypes and the resulting Voronoi tessellation. The experimental assessment shows that, in an ideal setting where the data distribution is known, the proposed approach is more robust to different data standardizations compared to the original one, based on a halfspace gating mechanism, and shows improved predictive performance.

Finally, we propose an adaptive mechanism for the choice of prototypes, which expands and shrinks the set of prototypes in an online fashion, making the model suitable for practical continual learning applications. The experimental results show that the adaptive model performances are close to the ideal scenario where prototypes are directly sampled from the data distribution.

Acknowledgements

I would like to express my gratitude to my supervisor, Professor Nicolò Navarin, who guided me throughout the development of this work, and to *Luca Pasa* and *Daniele Zambon*, whose help has been essential in these last months.

A special thank to *Giulia* who supports me in every challenge of life and is always by my side.

I would also like to thank my friends *Alberto*, *Matteo*, *Giulia* and *Melissa*, for the wonderful moments we spent, and continue to spend, together.

Finally, a big thank to my colleague and friend *Giulio* for the many discussions and coffee breaks, which I really needed.

Padova, Aprile 2022 Matteo Munari

Contents

1 Introduction **1**

2 Background **5**

2.1 Machine Learning 5
2.1.1 Definition 5
2.1.2 Settings 6
2.1.3 Training a machine learning model 6
2.1.4 Neural Networks 7
2.2 Continual Learning 9
2.2.1 Definition 10
2.2.2 Similarities with other machine learning paradigms 10
2.2.3 Task-aware vs Task-agnostic scenario 11
2.2.4 Task-aware Continual Learning Strategies 11
2.2.5 Task-agnostic Continual Learning Strategies 15

3 Gated Linear Networks **17**

3.1 GLN description 17
3.1.1 Geometric mixing 18
3.1.2 Gated geometric Mixing 19
3.1.3 GLN formulation 20
3.1.4 Training procedure of GLNs 20
3.2 GLN analysis 21

4 Prototypes Gated Linear Networks **25**

4.1 Prototype-based gating mechanism 25
4.2 Prototypes initialization 26
4.3 Growing ProtoGLN 27
4.3.1 Growth mechanism 28
4.3.2 GrowPGLN Formulation 32
4.3.3 Removal mechanism 33

5 Experiments **35**

5.1 Experimental setting 35
5.1.1 Models 35

5.1.2 Hyperparameters 35
5.1.3 Benchmarks 36
5.1.4 Evaluation 38
5.2 Context standardization on GLNs 41
5.3 Prototypes GLN 44
5.3.1 Data-driven prototypes initialization 45
5.3.2 Random prototypes initialization 49
5.4 GrowPGLN 49
5.4.1 Results with growth mechanism only 50
5.4.2 Results with both growth and removal mechanisms 54
5.5 Models comparison 56

6 Conclusions **61**

Abbreviations **63**

Bibliography **65**

List of Figures

3.1 Training procedure of GLN. The target is directly propagated to all the neurons, which compute a local gradient to update the weights. 21

3.2 Example of different region split. The same 2D dataset (black dots) are split into seven different regions by the red hyperplanes (left), while the blue hyperplanes are not able to separate the data, generating only a single non-empty region (right). 22

4.1 Region split in 2D produced by of halfspace gating (left) and gating with prototypes (right). The split produced by prototypes corresponds to a Voronoi Tessellation. 26

4.2 Example of the importance of the distance threshold d. When the value of d is too low (left), almost any new sample becomes a prototype, while with a high value of d (right) no new prototype is added. Only with the right value of d (center), a fair number of prototypes can be added. 28

4.3 Example of a neuron's prototype before (left) and after (right) the removal of the prototype in the middle. When a prototype is removed, the corresponding region is split and incorporated in the neighbors regions. After the removal, samples falling into the red shaded area are assigned to the closest of the remaining prototypes. 33

5.1 Example of a single task samples of Permuted MNIST. Each MNIST digit (upper images) is transformed using a random pixel permutation, obtaining a different image (lower images). The permuted images are meaningless for humans, but a neural network should be able to find the characteristic pattern of a digit, independently of the pixels relative position. 37

5.2 Task division in Split MNIST. Each task consist in a binary classification problem between two digits. The goal is to assign even digits to class 0 and odd digits to class 1. 38

5.3 Permuted MNIST dataset 2D representation using UMAP. The samples are colored by task. Samples of the same task are similar to each other and there is a sharp distinction between different tasks. 39
5.4 Split MNIST dataset 2D representation using UMAP. The samples are colored by task. There is no sharp distinction between samples of different tasks. 40
5.5 Comparison between different methods for data standardization on Permuted MNIST using GLNs. Shaded areas correspond to +/- one standard deviation. 41
5.6 Comparison between different methods for data standardization on Split MNIST using GLNs. Shaded areas correspond to +/- one standard deviation. 44
5.7 Comparison between different methods for data standardization on Permuted MNIST using ProtoGLN (data-driven initialization). Shaded areas correspond to +/- one standard deviation. . 46
5.8 Comparison between different methods for data standardization on Permuted MNIST using ProtoGLN (data-driven initialization). Shaded areas correspond to +/- one standard deviation. (magnification) 46
5.9 Comparison between different methods for data standardization on Split MNIST using ProtoGLN (data-driven initialization). Shaded areas correspond to +/- one standard deviation. 47
5.10 Comparison between different methods for data standardization on Split MNIST using ProtoGLN (data-driven initialization). Shaded areas correspond to +/- one standard deviation. (magnification) 47
5.11 Comparison between different methods for data standardization on Permuted MNIST using ProtoGLN (random initialization). Shaded areas correspond to +/- one standard deviation. 49
5.12 Comparison between different methods for data standardization on Split MNIST using ProtoGLN (random initialization). Shaded areas correspond to +/- one standard deviation. 50
5.13 Comparison between ProtoGLN and GrowPGLN with different dropout values on Permuted MNIST. All the results obtained with GrowPGLN are between the upper bound (green) of data-driven ProtoGLN and the lower bounds (red full and dotted lines) of random ProtoGLN. Shaded areas correspond to +/- one standard deviation. 51
5.14 Comparison between ProtoGLN and GrowPGLN with different dropout values on Split MNIST. The results obtained by GrowPGLN with dropout values set to 0.5 and 0.7 are between the upper bound (green) of data-driven ProtoGLN and the lower bounds (red full and dotted lines) of random ProtoGLN. Instead, GrowPGLN with dropout set to 0.3 presents a drop on the first task. Shaded areas correspond to +/- one standard deviation. . . 52

5.15 Comparison between ProtoGLN and GrowPGLN with removal mechanism on Permuted MNIST. The dropout rate of GrowPGLN is fixed to 0.5. The results obtained with GrowPGLN are between the upper bound (green) of data-driven ProtoGLN and the lower bounds (red full and dotted lines) of random ProtoGLN. Shaded areas correspond to +/- one standard deviation. 55
5.16 Comparison between ProtoGLN and GrowPGLN with removal mechanism on Split MNIST. The dropout rate of GrowPGLN is fixed to 0.7. The results obtained with GrowPGLN are between the upper bound (green) of data-driven ProtoGLN and the lower bounds (red full and dotted lines) of random ProtoGLN. Shaded areas correspond to +/- one standard deviation. 56
5.17 Comparison between GrowPGLN with (orange) and without (blue) removal mechanism on Permuted MNIST. The dropout rate of the GrowPGLN models is fixed to 0.5. The results obtained are almost identical. Shaded areas correspond to +/- one standard deviation. 57
5.18 Comparison between GrowPGLN with (orange) and without (blue) removal mechanism on Split MNIST. The dropout rate of the GrowPGLN models is fixed to 0.7. The results obtained are almost identical, with a slight difference in the accuracy on the last task. Shaded areas correspond to +/- one standard deviation. . . 58

List of Tables

5.1 Continual Learning Benchmarks 36

5.2 Example of accuracy matrix 40

5.3 Accuracy matrix (%) on Permuted MNIST with GLNs (feature-wise online standardization). 43

5.4 Accuracy matrix (%) on Split MNIST with GLNs (feature-wise offline standardization). 45

5.5 Accuracy matrix (%) on Permuted MNIST with data-driven ProtoGLN (feature-wise online standardization). 48

5.6 Accuracy matrix (%) on Permuted MNIST with GrowPGLN (dropout rate = 0.5). 53

5.7 Accuracy matrix (%) on Split MNIST with GrowPGLN (dropout rate = 0.7). 54

5.8 Accuracy matrix (%) on Split MNIST with GrowPGLN (dropout rate = 0.3). 54

5.9 Accuracy matrix (%) on Split MNIST with GrowPGLN (dropout rate = 0.5). 55

5.10 Average percentage accuracy (A) of all tested models. The upper part of the table reports the results relative to the methods that can be directly applied to a continual learning scenario. The lower part reports the models that are not suited for real continual learning scenarios, but that are useful for the comparison. The results show that GrowPGLN achieve the best average accuracy on both datasets between the practical solutions. The model performances are also not so far from the upper bound of ProtoGLN. 58

5.11 Combination of accuracy matrix on Permuted MNIST of the three best models directly applicable to continual learning scenarios: GLN with online standardization, ProtoGLN (Proto) with online standardization, GrowPGLN (Grow) with no standardization (dropout = 0.5). Each cell reports the accuracy obtained by GLN (top), ProtoGLN (middle) and GrowPGLN (bottom), while each column shows the variation of the models' accuracies throughout the training process. GrowPGLN achieves a better accuracy in every task at every step of training . The best results of each cell are highlighted in bold. 59
5.12 Combination of accuracy matrix on Split MNIST of the three best models directly applicable to continual learning scenarios: GLN with online standardization, ProtoGLN (Proto) with online standardization, GrowPGLN (Grow) with no standardization (dropout = 0.7). Each cell reports the accuracy obtained by GLN (top), ProtoGLN (middle) and GrowPGLN (bottom), while each column shows the variation of the models' accuracies throughout the training process. GrowPGLN achieves a better accuracy in every task at every step of training . The best results of each cell are highlighted in bold. 60

Chapter 1

Introduction

The idea of building artificial human-like creatures has always fascinated humanity, just think about the ancient Greek myth of Talos, a giant bronze automaton built to defend Crete from invasions by sea, or Leonardo da Vinci's studies on automata during the Renaissance.

From the 16th century, the development of Science, in particular Physics, Mechanics and Electronics, led to machines able to ease, substitute and improve human labor in almost any practical activity. Yet, those first machines were far from being defined human-like, because even the simplest ability of a child was out of their capabilities.

The breakthrough came in the 20th century, with the born of electronic computers and Computer Science. Computers can calculate any computable function, as proved by Turing [45], thus if we reduce each human action to just a reaction to some stimuli, it is straightforward to see that with the right programming, a computer can reproduce any human behavior. However, manually defining programs capable of reproducing complex behaviors like talking and reasoning is really hard, because we do not have a deep understanding of how language and thoughts are produced by brain.

Nowadays, Machine Learning and Deep Learning seem to be the best way to tackle such complex problems. These methods are designed to fill the gaps in our knowledge exploiting data and extracting knowledge from them, i.e. they use data to *learn*. One of the most popular machine learning models are Neural Networks, because, since their revival in the 2010s, they proved themselves to be astonishingly good and capable of outperforming humans in a variety of task, e.g object recognition, translation or playing games.

However, there is still a huge gap between neural networks and humans. This is due mainly to the lack of adaptation capabilities of the current models and to the differences in the learning process.

A human being learns in an incremental way for its entire lifetime, continually adapting to new conditions and learning new abilities when required. Instead, traditional machine learning models are often trained for a finite amount of time to solve a single or few tasks.

A recent machine learning paradigm, called Continual Learning, focuses on the development of machine learning models that are trained in a more human-like fashion. In fact, continual learning assumes that the data distribution and task to solve can change with time and that the learning process is potentially infinite.

Typically, learning in time-varying environments is carried out by training a model on available data and, as new data is observed, the learned model is updated according to an adaptation mechanism. Several hard challenges about learning from streaming data have been recently discussed [4]. Among all, the inability of retaining relevant knowledge from past tasks –also known as forgetting– stands out as a major issue [33].

In this work we propose a new continual learning method based on a recently proposed neural model called Gated Linear Network (GLN) [6, 46]. We first study the problem of forgetting associated with GLNs. GLNs are deep neural networks that exhibit an important advantage for addressing continual learning applications. Differently from the majority of modern neural models, GLNs can be efficiently trained without backpropagation, making them, in a broad sense, more biologically plausible than other approaches [8]. Instead, GLNs employ a gating mechanism at the neuron level that results in local learning rules based on easy-to-solve quadratic optimization problems. In turn, this enables the independent training of each GLN neuron. Compared to backpropagation, this approach results more parallelizable since, after the forward pass, each neuron can be updated separately.

We collect empirical evidence to understand what factors cause model forgetting, and how it is possible to mitigate this problem. Firstly, we study the problem of data standardization in GLNs and what is its impact on forgetting. We discovered a strong relation between the performance of GLN and the adopted data standardization approach. In particular, the initialization of hyperplanes used by the gating mechanism should be gauged with the data distribution and preprocessing.

Finding a suitable combination between data standardization and neuron gating is not straightforward, since that data distribution is allowed to change over time. We collect empirical evidence that the choice of standardization applied to the data highly influences the forgetting of GLNs.

Secondly, we note that knowing in advance the data distribution to properly calibrate the data standardization and the gating mechanism is often an unrealistic assumption to make. Even estimating the data distribution through acquired data is unreasonable in a continual learning setup.

To mitigate this problem, we propose an alternative gating definition based on prototypes instead of hyperplanes. This variation brings several benefits, including an easier way to incorporate the knowledge about the support of the data distribution. We show that, when the prototypes provide a reasonably good coverage of the data, then we obtain higher performance compared to the original formulation. We also show that this approach is independent of the adopted data standardization.

Thirdly, we introduce an effective mechanism for the selection of prototypes.

This mechanism do not require to know in advance the data distribution, in agreement with continual learning assumptions.

Finally, we empirically demonstrate that the proposed methods displays less forgetting.

This dissertation is structured as follows. Chapter 2 introduces the background and related works. In particular, it gives an overview of the Machine Learning field and compares it with the Continual Learning framework. It also covers all the most important aspects of continual learning, going from a general definition to the state-of-the-art techniques. Chapter 3 describes the Gated Linear Networks model, which this work is based on. It also analyzes the problems of GLNs, specially the relation between gating mechanism and data standardization. Chapter 4 presents Prototypes Gated Linear Networks, our first variation of GLNs. It introduces also a growing version of Prototypes GLN, directly applicable to continual learning scenarios. Chapter 5 reports and discusses the experimental results, comparing our models to the original GLN. Finally, Chapter 6 summarize the contribution of this work and discuss possible future improvements.

Chapter 2

Background

This chapter presents the knowledge background necessary to understand our contribution.

Starting from a general description of the Machine Learning field (Sec. 2.1), we introduce the Continual Learning framework (Sec. 2.2), with its goals and challenges.

We also present some models developed for continual learning in Section 2.2.4.

2.1 Machine Learning

Machine learning is a wide branch of the Artificial Intelligence field, whose general goal is the development of artificial intelligent agents. There is not a formal definition of intelligence, and the possibility of reproducing it artificially is still discussed today. However, the general idea is that an intelligent machine or program should be able to behave like a human, solving the same tasks a human being is able to do, e.g. talking, listening, reading, walking etc.

The difference between machine learning and other artificial intelligence approach resides in the way this goal is pursued.

At the basis of machine learning there is a *learning* process, meaning that a program should use some external knowledge to learn a particular ability.

The next sections give an overview of the machine learning field, with a particular focus on neural networks.

2.1.1 Definition

We report the classical definition by Thomas Mitchel [34], stating:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E".

In other words, such a program have to use a set of samples, called *dataset*, to build a mapping from the input it receives to the correct output, according to the task to solve. A machine learning dataset is usually made up of two parts: a training set Tr, used in the training phase, and a test set Te, used to evaluate the performance of the program in solving the task T.

The mapping from input to output is just a mathematical function, which associate the correct output to each input received by the program. Usually, this function, also called *target function*, is unknown and really hard to define manually due to its complexity, that is why it is learned from the data.

The aim of machine learning, thus, is to develop models that approximate the target function that can be trained using data.

2.1.2 Settings

There is a high number of tasks that can be solved with machine learning, like regression, machine translation, denoising etc.

Depending on the type of tasks to solve and the data available, we have four main machine learning settings:

- **Supervised learning**: the dataset is made by a set of (x, y) pairs, where x is a sample and y is the correct output, called *target*. The knowledge of target can be exploited to correct the model when it outputs a wrong result. This setting is called supervised because it is as the correct target is given by an external expert or supervisor.

- **Unsupervised learning**: the dataset is composed by just the examples x, with no additional information. This type of dataset are usually used to find patterns, groups or estimate the example distribution.

- **Semi-supervised learning**: only a small part of the dataset is labeled, i.e. presents a target. This setting is a combination of the supervised and unsupervised settings.

- **Reinforcement learning**: this setting is quite different from the previous ones. In reinforcement learning tasks, the target function maps a environment state s to an action a that an agent can perform. After performing the chosen action, the agent ends in a new state s' and receives a reward r, which can be a positive or negative real number, depending on the state s'.

In this work, we focus on supervised classification tasks.

2.1.3 Training a machine learning model

In a supervised learning setting, we want to approximate a function f, such that $f(x) = y$, where x is a sample and y the relative target. From a probabilistic point of view, this is equivalent to use a finite set of (x, y) pairs to estimate

the probability $p(y|x)$. The goal is to use the estimated probability to perform prediction over unseen data.

This is slightly different in a unsupervised setting, where the goal is to estimate the samples distribution $p(x)$, which, for example, can be later used to generate new samples or to detect outliers.

Estimating a probability from a set of samples to perform prediction on unseen data works if we assume two conditions to be true. First, the data distribution has to be stationary, meaning that do not change in time. In this way, the estimated probabilities approximate the true distribution also when the training is ended.

Second, the dataset has to be a representative sample of the data distribution. Since usually the dataset is entirely available from the beginning, we can assume the data as independent and identically distributed (i.i.d.).

In general, these assumptions hold when we focus on a single task and the dataset is big enough. However, in more complex scenarios, we need different training approaches, as discussed in Section 2.2.

2.1.4 Neural Networks

Though many type of machine learning models and algorithms have been developed through the years (e.g. SVM [5], Decision Trees [38], Bayesian methods, Clustering methods), Neural Networks are the models that in recent years have gained more attention, because of their astonishing results in solving complex task like object recognition or translation.

Neural Networks are machine learning models inspired by the human brain structure. As human brain is made up of billions of interconnected neurons, neural networks are defined by the composition of smaller units. These units are called artificial neurons or just neurons.

The first artificial neuron, the Perceptron, was introduced in [39]. The Perceptron is a mathematical abstraction of a human neuron, firing or not firing depending on its input, and it is defined as

$$o = f(\mathbf{w}^\top \mathbf{x})$$

where $\mathbf{x} \in \mathbb{R}^n$ is the input vector representing a training sample, $\mathbf{w} \in \mathbb{R}^n$ is a vector of learnable weights and $f(\cdot)$ is a threshold function defined as

$$f(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{otherwise} \end{cases}$$

.

The original formulation adds also a bias term b, here omitted for simplicity, because it can be incorporated into the weights defining $\mathbf{w} = [w_1, ..., w_n, b]^\top$ and $\mathbf{x} = [x_1, ..., x_n, 1]^\top$.

Multiple neurons can be combined into layers of the form

$$\mathbf{o} = f(\mathbf{W}\mathbf{x})$$

where $\mathbf{W}$ is a weights matrix, with each row corresponding to a single neuron's weights vector, and $f(\cdot)$ is applied element-wise. In a single layer neurons work in parallel.

If we stack two layers of neurons, we obtain a Multi-layer Perceptron, where the output of the first layer is fed into the second layer as

$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x})$$
$$\mathbf{o} = f(\mathbf{W}^{(2)}\mathbf{h})$$

where $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are the weights matrix of the first and second layer respectively. We denote the first layer output as $\mathbf{h}$, because internal output are *hidden*, since they are not visible outside the network. The $g(\cdot)$ function, called activation function, must be a non-linear function. Otherwise, the composition of layers would be equivalent to a single layer where we have a weights matrix $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$.

It has been proved [16, 9] that any Multi-Layer Perceptron with at least a hidden layer and a sufficient number of hidden neurons is a universal function approximator, meaning that for any continuous function, it exists a set of weights such that the network exactly represent that function. However, the number of required neurons could be huge and there is no guarantee that a learning algorithm would be able to compute those weights.

In [12], the authors argue that a deeper network architecture, i.e. with more than two layers, can exponentially reduce the number of required neurons, for some families of problems.

Thus, the basic neural network structure consist in stacking multiple layers. An L-layer neural network can be expressed by the following recursive formulation

$$\begin{cases} \mathbf{h}^{(0)} = \mathbf{x} \\ \mathbf{h}^{(l)} = g\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)}\right), \quad l = 1, \dots, L. \end{cases} \tag{2.1}$$

where $\mathbf{h}^{(L)} = \mathbf{o}$

The type of neural networks defined by equation (2.1) are called *Feedforward* Neural Networks for their hierarchical architecture that makes the computation flow going forward from the first to the last layer. There are many other networks types, like Convolutional Neural Networks or Recurrent Neural Networks, which have other characteristics, but they are not object of this work. Thus, when we use the term neural networks we refer to feedforward neural networks.

Training Neural Networks

Neural networks, as any machine learning model, depends on some parameters that can be updated during the training process. In neural networks, these parameters are the network's weights, which are used to compute the output.

The goal of the training is to find the best value of the weights, minimizing a loss function J, which measures the error done by the network.

This is done via gradient descent, an first-order optimization technique that iteratively update the network's weights searching for a minimum of the loss function. In fact, the loss function is unknown, thus, its minima can not be determined analytically.

The gradient descent update formula is defined as

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} J$$

where η is a scaling factor called learning rate and $\nabla_{\mathbf{w}} J$ is the gradient of the loss function J w.r.t. the vector of weights $\mathbf{w}$. On each update step, the gradient ∇ is calculated from the entire dataset, i.e. summing the gradients of each sample.

As its name suggest, gradient descent exploits the gradient of the loss function to determine the best weights update. With a single layer it is easy to compute the gradient w.r.t. the vector of weights $\mathbf{w}$, but with a deeper network the gradient computation is not straightforward. The computation of a deep network gradient is done using *backpropagation* [40], which is based on the chain rule of calculus and consist in splitting the gradient computation into steps, starting from the final layer and going backward down to the first layer. This mechanism, though very effective, can be very slow, because its complexity is linear in the network depth. Moreover, the gradient descent algorithm usually needs multiple steps to converge to a solution, also re-using the same data multiple times.

Usually, to reduce the computational time, the Stochastic Gradient Descent (SGD) method is used. In SGD, the gradient is computed on a subset of data and not on the entire dataset.

2.2 Continual Learning

Continual learning extends the classical machine learning framework presented in Section 2.1. The focus of machine learning is usually on solving a single task. In continual learning, instead, the learning process continues for an indefinite period of time, in an environment where the data distribution and the task to be solved might change over time [35]. Thus, there are three main differences:

- The learning process is potentially infinite. This means that a continual learning model should be designed to ideally work forever. Moreover, this revolutionizes the way a model's performances are evaluated, because there is not a final moment where we can assume the model to be completely trained.

- The data distribution cannot be assumed stationary, meaning that the target function that the model has to approximate is not always the same and depends on time. For example, there could be a shift in the input distribution, when completely new samples are observed, or even in the target distribution, meaning that old sample are associated to a different output in different moments.

- The task to be solved might change over time. In this case the model has to deal with multiple tasks, thus, it should be able to reuse old knowledge and adapt to different environments.

In the next sections, we see how these relaxations of the usual machine learning process assumptions deeply affects the ability of machine learning models, in particular of neural networks.

2.2.1 Definition

In [25], Mitchell's machine learning definition is adapted to the continual learning framework as:

"A computer program is said to learn continually from experience if, given a sequence of ephemeral partial experience E_i, a target function h^* and performance measure P, its performance in approximating h^* as measured by P improves with the number of processed partial experience E_i."

The definition exposes a crucial aspect of continual learning: the data are *ephemeral* and they are not all available from the beginning. This means that the data distribution and/or the target distribution might change over time.

One of the most relevant issues of learning in a non-stationary environment is that the neural networks tends to adapt to the current data distribution, without retaining past knowledge.

This phenomenon is known as *catastrophic forgetting*, or simply forgetting, because it is like the network forgets how to solve old tasks. Forgetting on neural networks has been widely studied in literature [35, 36, 19, 25].

To better define catastrophic forgetting, consider a model already trained to solve a task T_i. Catastrophic forgetting occurs whenever there is a significant drop of the model performances in solving task T_i, after learning a new task T_j.

In neural networks, forgetting occurs when new data samples are significantly different from past data, thus when there is a change in the data/target distribution or in the task to be solved. The network keeps updating its weights according to the new data, possibly compromising past knowledge. This leads to a drop in performance on past tasks or over data coming from the initial distribution. This *naïve* approach, consisting in just fine-tuning the network with new data, does not apply any particular strategy to prevent forgetting. Thus, it usually assumed as a lower bound on the model performances.

In Section 2.2.4, we present a variety of strategies designed to tackle catastrophic forgetting.

2.2.2 Similarities with other machine learning paradigms

Continual learning presents many common aspects with other machine learning paradigms, the most similar being online learning [23, 36].

In the online learning setting, we assume that the data is available sequentially. Also, the data distribution might change over time, as it happens in continual learning. However, the focus of online learning is on the promptness

of training, because models have to be updated frequently to keep pace with the changing data distribution. Moreover, usually, in online learning we assume that data from the past distribution will not be seen again. Thus, the problem of catastrophic forgetting is not relevant in this setting, because the model is evaluated according to the current data distribution and we are not interested on the performance over past data.

Another paradigm related to continual learning is multi-task learning [23]. In this case the goal is to train a model able to solve multiple task at the same time, feature required also in continual learning. In multi-task learning, though, the data of different task is available from the beginning.

Also the transfer learning setting [23] is similar to continual learning: they both update a pre-trained model to solve a new different task. Still, in transfer learning there is no interest in preserving the original performances of the pre-trained model and catastrophic forgetting is not considered a problem.

2.2.3 Task-aware vs Task-agnostic scenario

An important feature that strongly influences the performance of a model in continual learning is the knowledge of the task to be solved.

The majority of the first developed continual learning methods, presented in Section 2.2.4, assumes to know the task label at training time and some of them even at test time. This scenario is commonly called task-aware, meaning that the model knows the task it is going to solve. Basically, in this setting the model receives a task label T_i in addition to the samples. Thus, it can exploit that information to avoid forgetting, for example implementing a knowledge consolidation mechanism when a task switch occurs.

The opposite scenario is called task-agnostic, meaning that the model does not get the task label information neither at test time, nor at training time. The task-agnostic scenario is more general, but also more challenging, because developed models have to be robust to task changes without knowing when they will happen.

This type of methods is presented in Section 2.2.5.

2.2.4 Task-aware Continual Learning Strategies

As explained in Sec. 2.2.1, continually updating a model with a naïve approach leads to catastrophic forgetting.

The *Cumulative* approach is the simplest solution to try to solve this problem. It consists in retraining the model from scratch when the task changes, accumulating all past data and reusing them for the training. It is evident that this method have two strong limitations: memory consumption and computational complexity. The amount of data to be stored could be huge and the training time grows linearly with the number of samples. For some applications this could be sustainable, but in continual learning the training process is ideally infinite, thus, at some point, the high memory requirements and computational time would become a real issue.

Thus, many methods have been proposed to face catastrophic forgetting. They are usually divided into categories, depending on the adopted strategy. In literature [28, 25], four main categories have been identified: architectural, regularization, rehearsal and generative replay. There is also a fifth category of hybrid methods, which comprehends those methods that combine different approaches, since the different strategies are not mutually exclusive.

Each strategy is explained in detail in the following subsections, providing some example of proposed methods.

Architectural methods

The main feature of architectural methods is the definition of functional areas inside the network. The idea comes from the human neocortex-hippocampus duality, as theorized by the Complementary Learning Systems theory [32]. According to this theory, the hippocampus is able to learn fast, storing and recognizing short-term episodic memories, while the neocortex is slowly trained to recognize general patterns.

For instance, this is the approach Copy-Weights with Re-init [26] is based on. This method, designed to solve image classification tasks, consist in defining two sets of weights as last layer of a convolutional neural network: a set of consolidated weights *cw* and a set of temporary weights *tw*. The lower layers of the networks are pre-trained and fixed. In CWR, the training is done only on the temporary weights, which are re-initialized whenever a shift in the data distribution occurs. Before the re-initialization, the temporary weights *tw* are used to update the consolidated weights.

Of course, the number of distinct areas can be much larger than two. For instance, Progressive Neural Networks [41] instantiates task-related neurons every time a new task is encountered, thus the network architecture is expanded dynamically. This leads to task-related areas of the network, that are trained and selected only using samples of the related task, by freezing the other parameters. In this way, the weights related to old task can be preserved from changing.

Another architectural approach is the one adopted by Context dependent Gating (XdG) [30]. This method uses the task label as *contextual information*, defining a task-related random mask that is used to deactivate a subset of the neurons, obtaining a sparse network. In this way, each neuron is trained only on a few tasks, the ones where it is not masked, reducing the probability of forgetting.

Regularization-based methods

A regularization-based method consists in applying a regularization term to the loss, such that the learning of new tasks do not interfere with the previously learned ones.

The loss function can be modified in many ways. For example, Learning without Forgetting [24] exploits an extra loss term, called Knowledge Distillation loss [15], which has the effect of penalizing strong variation of the network predictions. To do so, for each new sample $\mathbf{x}$ the old model is used to predict the output y_{old}, as

$$y_{old} \leftarrow predict(\mathbf{x})$$

These outputs are then used to compute the knowledge distillation loss

$$\mathcal{L}_{KD}(y_{old}, y_{new})$$

where y_{new} is the output of the updated model.

Another approach consists in penalizing the change of important parameters. Lots of different methods are based on this idea, each one proposing different ways to compute the parameter's importance. For example, Elastic Weights Consolidation [19] uses the diagonal of the Fisher information matrix F computed on the old tasks parameters. In EWC, the loss is defined as

$$\mathcal{L} = \mathcal{L}_{new} + \sum_i \frac{\lambda}{2} F_i (w_i - w_{i_old})$$

where $\mathcal{L}_{new}$ is the loss term on the new task, λ set how important the old tasks are w.r.t. the new one, the w_i are the updated weights and the w_{i_old} are the old weights.

Synaptic Intelligence (SI) [49] and Memory Aware Synapses (MAS) [1] have a similar formulation for the loss

$$\mathcal{L} = \mathcal{L}_{new} + \lambda \sum_i \Omega_i (w_i - w_{i_old})$$

but the parameters importance Ω is computed using the loss curvature w.r.t. the parameters in SI, and using the learned function curvature w.r.t. the parameters in MAS. The parameters Ω are recomputed whenever a task-switch occurs.

Another regularization method is Uncertainty-guided Continual learning with Bayesian Neural Networks (UCB) [10]. In UCB each weight w is drawn from a normal distribution $w \sim \mathcal{N}(\mu, \sigma^2)$. Hence, during training, we do not learn the exact value of w, but we estimate the parameters of the normal distribution it is sampled from. In this case the parameters importance is given by the parameter's uncertainty σ^2. Intuitively, weights with low uncertainty (low variance) are specialized in solving a particular task, while weights with high variance do not influence the network's output. The parameter importance in this case is not used to penalize the loss computation, but to scale the learning rate η associated to the parameter, reducing changes to certain weights while not limiting uncertain weights. At each training step, the learning rate is updated as

$$\eta_i \leftarrow \eta_i \cdot \sigma_i$$

hence low values of σ shrinks the learning rate, reducing the update delta, while bigger values might increase it, with the opposite effect.

Rehearsal methods

Rehearsal methods are based on the idea of retraining the network using past data. The cumulative approach presented in Section 2.2.4 is an example of full-rehearsal, i.e. all past data are kept and used to retrain the model, and its limitation have already been discussed.

Hence, rehearsal methods are usually defined to select and store only the most relevant samples, avoiding the issues of the cumulative approach. An example of rehearsal method is ExStream [14].

In ExStream, a buffer is used to store class-related samples, which can be considered as a set of prototypes. Each new sample is stored into the buffer, until this one is full. When it is not possible to insert new data, the two most similar prototypes $\mathbf{p}_i$ and $\mathbf{p}_j$ are merged, freeing up space. The merge is done based on the number of times those prototypes have been merged before, as

$$\mathbf{p} \leftarrow \frac{n_i \mathbf{p}_i + n_j \mathbf{p}_j}{n_i + n_j}$$

$$n \leftarrow n_i + n_j$$

where n_i and n_j is the number of sample a prototype represents.

Generative replay methods

The Generative Replay category is related to the previous one, because the approach used to mitigate the catastrophic forgetting is the same. As for rehearsal methods, the model is retrained on both new and old tasks. The difference resides in the samples used to retrain the network. While rehearsal methods just store old samples, generative replay methods are able to generate data coming from the learned distribution. In this way there is no need to store past data.

Moreover, Generative methods are versatile, because usually they can deal with both supervised and unsupervised tasks.

An example of generative method is Generative Replay [43]. In GR, the network is made of two distinct components: a generator and a solver. The generator is trained to approximate the data distribution, while the solver learns how to solve the task.

The fundamental aspect of GR is that each component is trained using both the training data and the data generated by the generator. In this way, the model is trained from data coming from both the current and past data distributions.

Hybrid methods

The four strategies described in the previous sections do not exclude each other. In fact, it is possible to combine different strategies.

For instance, AR1 [28] extends CWR, an architectural method, with SI, a regularization method. In particular, instead of freezing most of the network parameters as CWR does, they are trained with the same approach of Synaptic Intelligence, hence penalizing variations of the most important parameters.

2.2.5 Task-agnostic Continual Learning Strategies

As discussed in Section 2.2.3, the task-agnostic scenario is the most challenging, because the model do not know when a task switch occurs.

There are two main approaches in dealing with a task-agnostic scenarios. The first type of approach consist in actively estimating the current task or, alternatively, the time when a task switch happens.

The other type of approaches does not explicitly predict the current task. In fact, they usually assume to learn a single complex task and that the only changes occur in the data and/or target distribution.

We list below some example of task-agnostic methods, grouping them in the same categories introduced in Section 2.2.4.

Architectural

Two architectural task-agnostic methods are Gated Linear Network [46] and Dendritic Gated Network [42].

The GLNs network architecture is similar to a feedforward network, with a hierarchical structure of neurons. However, differently from standard neurons, a GLN neuron presents multiple weights vectors. For each new sample, the weights to be used are selected based on some external information, or context. Hence, the network architecture changes depending on the environment conditions. The idea is that the context brings information about the task to solve, thus it can be used to configure the network via a gating mechanism, that for each neuron selects context-related weights. Being the starting point of this work, GLNs are presented in detail in Chapter 3

DGN, instead, is an extension of GLNs, which tries to make the gating mechanism of GLNs more similar to the inhibitory mechanism of the neuron synapses. In DGN, each neuron presents a set of *branches*, that, like the dendrites of biological neurons, are linked to different subset of neurons of the lower layer. In this case, the context is used to activate or deactivate each of the neuron's branches. Hence, the gating mechanism is not applied to the weights, but to the neuron's input.

Regularization

Task-Free Continual Learning [2] extends Memory Aware Synapses to the task-agnostic case introducing a task-switch detection mechanism. The method monitors the loss function and detects a task change when there is a quick growth of the loss. The assumption is that when the model performs badly on new data, it means that the task has changed.

This information is then used to decide when to update the weights importance Ω defined in MAS.

If the loss is stable w.r.t. the last few training steps, then the model is learning the same task and the value of Ω can be safely updated. Instead, if the loss presents a sudden peak, the task has probably changed and the parameters Ω are fixed.

Hybrid

Most of task-agnostic methods are hybrid, which means that they combine different strategies. For example, Hybrid generative-discriminative approach to Continual Learning (HCL) [18] presents a dynamic architecture, because it can expand the model when there is a drift in the data distribution. The model is based on normalizing flows, thus it can be used to predict the task probability and to detect a task-switch. Then, to train the normalizing flows, it can either use a generative replay approach or regularization techniques.

Continual Neural Dirichlet Process Mixture (CN-DPM) [22] is based on a similar idea. The model is composed by many components, which estimate the data distribution to detect outliers, that are temporarily stored in a buffer. Outliers are considered to come from a new task, thus, when the buffer is full, the samples are used to expand the network and train the new added component. Instead, for non-outlier data, each sample is used to update the network using a generative replay approach.

The same idea and training mechanism of CN-DPM is present in Continual Unsupervised Representation Learning, with the only difference that the component is implemented by a Variational Autoencoder [17].

Finally, Dark Experience Replay [7] combines a regularization and a rehearsal mechanism, applying a knowledge distillation loss, like Learning without Forgetting, computed on a buffer of past samples. The stored samples are selected via reservoir sampling, a technique that guaranties that the buffer content is equivalent to a set of samples drawn uniformly from the past data.

Chapter 3

Gated Linear Networks

In this chapter we discuss in more detail the Gated Linear Network model. Section 3.1 exhaustively describes the functioning of GLNs, while in Section 3.2 we analyze the limitations of the halfspace gating mechanism in a continual learning setting.

3.1 GLN description

Gated Linear Networks (GLN) [46] are similar to neural networks, but their functioning is pretty different.

In neural networks, each layer basically learns to recognize specific patterns in the data. The pattern complexity grows with the level of the layer, until we reach the final layer which uses the last representation it receives in input to perform a prediction.

In GLNs, instead, each neuron is considered as an expert that directly predicts the probability of the target, combining and weighting the outputs of the experts (neurons) of the previous layer. In this way, the network corresponds to a hierarchical mixture of experts, where the top-level expert prediction is considered as the output of the network. Moreover, each neuron chooses the set of weights to use (and train) depending on the region of space the input data falls into, with each neuron partitioning the input space differently. Hence, the obtained regions changes from neuron to neuron.

From a high level perspective, the input-based weight selection can be interpreted as if an expert changes its way of weighting the prediction of other experts, based on same external information or context.

GLNs present multiple advantages, especially for continual learning. For example, each neuron of a GLN can be trained locally and independently –instead of exploiting backpropagation,— and its weights optimized according to a convex loss, as discussed in Section 3.1.4. The absence of backpropagation makes GLNs particularly adapt to a continual learning setting. In fact, the computational

complexity of an update step is reduced to constant time w.r.t. the network depth, while with backpropagation the time required for the computation of the gradient is linear.

Moreover, it has been shown that GLNs can approximate to arbitrary precision any function that represents a probability [47]. GLNs output values in [0, 1], which can be interpreted as probability measures.

These properties make GLNs suited for data-efficient batch learning, but also for online and continual learning tasks.

As usual with neural networks, GLNs neurons are organized by layers, in which the output of a layer becomes the input for the following one. However, differently from other kinds of neural networks, each neuron is trained locally. In fact, each neuron tries to approximate the target optimizing a local loss function. Thus, the global error signal is directly sent to every neuron of the network. Each neuron in a GLN can be described as a *Geometric Mixer*, presented in Sec. 3.1.1, on which it is applied a gating mechanism, like the one described in Section 3.1.2.

Although GLNs can be defined for both classification and regression problems, in this work we focus on the classification case.

3.1.1 Geometric mixing

A GLN's neuron, called Geometric Mixing Neuron, is a computational unit of the form

$$\mathbf{h} \mapsto \sigma\left(\mathbf{w}^\top \sigma^{-1}(\mathbf{h})\right), \tag{3.1}$$

where $\mathbf{h}$ is the input vector to the neuron, $\mathbf{w}$ a parameter vector, $\sigma(x) := 1/(1+e^{-x})$ is the sigmoid function, and $\sigma^{-1}(x)$ its inverse –the logit function, applied element-wise to vectors. We omit the bias terms for the sake of simplicity.

A neuron defined like in Eq. (3.1) assumes the components of the input vector $\mathbf{h}$ to be in $[0,1]$, and returns a value in $[0,1]$ as well, such that the output of a layer of neurons can be passed as input to the next layer neurons. Accordingly, an L-layer network of geometric mixing neurons can be defined, for a generic input data point $\mathbf{x}$, as

$$\begin{cases} \mathbf{h}^{(0)} = \sigma(\mathbf{x}) \\ \mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\sigma^{-1}(\mathbf{h}^{(l-1)})\right), \quad l = 1, \dots, L. \end{cases} \tag{3.2}$$

It is easy to show that a deep network of geometric mixing neurons like in Eq. (3.2) is equivalent to a linear network with a final sigmoid activation, i.e.,

$$\mathbf{h}^{(L)} = \sigma\left(\mathbf{W}^{(L)}\sigma^{-1}\left(\sigma \cdots \left(\mathbf{W}^{(1)}\sigma^{-1}\left(\sigma\left(\mathbf{x}\right)\right)\right)\right)\right) \tag{3.3}$$

$$= \sigma\left(\mathbf{W}^{(L)} \cdots \mathbf{W}^{(1)}\mathbf{x}\right) \tag{3.4}$$

$$= \sigma\left(\prod_{l=1}^{L} \mathbf{W}^{(l)}\mathbf{x}\right), \tag{3.5}$$

because the intermediate sigmoid function at layer l simplifies with the logit function at layer $l+1$. To render nonlinear functions, each neuron implements a *gating* mechanism.

3.1.2 Gated geometric Mixing

GLN neurons perform gating on side information, or contextual information, associated with the input data. We denote with $\mathbf{x} \in \mathbb{R}^{d^{(x)}}$ the input data, and with $\mathbf{z} \in \mathbb{R}^{d^{(z)}}$ context vectors representing the side information.

The gating of each GLN neuron consists in partitioning the context space $\mathbb{R}^{d^{(z)}}$ into non-overlapping regions. Each region is associated to a different parameter vector $\mathbf{w}$ of Eq. (3.1), resulting in the so-called *Gated* Geometric Mixing [31].

The association from contexts to regions is carried by a *region assignment* function c. The effect of gating is that data instances with contexts falling in the same region are processed by the same geometric mixer. The c function is defined depending on side-information $\mathbf{z}$ associated to each input, but, as mentioned before, usually $\mathbf{z} = \mathbf{x}$.

Halfspace gating

The original paper [46] proposes to implement the gating in the c functions with a half-space gating mechanism. Given a vector $\mathbf{z} \in \mathbb{R}^{d^{(z)}}$, and a hyperplane with parameters $\mathbf{a}_i \in \mathbb{R}^{d^{(z)}}$ and $b_i \in \mathbb{R}$, let us define a context function $\tilde{c}_i : \mathbb{R}^{d^{(z)}} \to \{0,1\}$ as:

$$\tilde{c}_i(\mathbf{z}) = \begin{cases} 1 & \text{if } \mathbf{a}_i^\top \mathbf{z} > b_i \\ 0 & \text{otherwise} \end{cases}$$

where hyperplane $\mathbf{a}_i^\top \mathbf{z} = b_i$ divides $\mathbb{R}^{d^{(z)}}$ into two half-spaces. Assuming the number k of regions to be a power of 2, we can stack $\log_2(k)$ context functions of the same kind, obtaining a higher-order context function $\tilde{c} : \mathbb{R}^{d^{(z)}} \to \{0,1\}^{\log_2(k)}$, i.e. $\tilde{c} = [\tilde{c}_1, \ldots, \tilde{c}_k]^\top$.

We can then easily define a function f mapping from $\{0,1\}^{\log_2(k)}$ to $\{0, \ldots, k-1\} \subset \mathbb{N}$, obtaining the function $\hat{c} : \mathbb{R}^{d^{(z)}} \to \{0, \ldots, k-1\}$, $\hat{c} = f \circ \tilde{c} = f(\tilde{c}(\mathbf{z}))$. We can exploit the one-hot encoding of the output of such function and re-define it as $c : \mathbb{R}^{d^{(z)}} \to \{0,1\}^k$, $c = \mathit{one_hot}(\hat{c})$.

Given a layer i, each neuron j computes a different function $c_j^{(i)} : \mathbb{R}^{d^{(z)}} \to \{0, \ldots, k-1\}$. For the j-th neuron at the i-th layer, the output of the context function applied to $\mathbf{z}$ is thus the (one-hot) vector $\mathbf{c}_{j,(\mathbf{z})}^{(i)}$.

Weight vector selection

Given the pair $(\mathbf{x}, \mathbf{z})$ as input, we can select the weights of a single neuron j at the i-th layer (i.e. the j-th row of $\mathbf{W}^{(i)}_{(\mathbf{z})}$) as:

$$\mathbf{w}^{(i)}_{j,(\mathbf{z})} = \mathbf{\Theta}^{(i)}_j \mathbf{c}^{(i)}_{j,(\mathbf{z})} \tag{3.6}$$

where $\mathbf{\Theta}^{(i)}_j \in \mathbb{R}^{d_{i-1} \times k}$, d_{i-1} is the number of neurons of layer $i-1$, k is the number of regions, and $\mathbf{c}^{(i)}_{j,(\mathbf{z})} \in \mathbb{R}^k$. In the formulation, we assume for the sake of simplicity that each neuron in the network considers the same number of regions. However, having a variable number of regions per neuron is possible as discussed for the model proposed in Section 4.3.

Notice that the main characteristic of a Gated Linear Neuron is that, instead of having a single weight vector, each GL neuron depends on a *matrix* of parameters $\mathbf{\Theta}^{(i)}_j$.

3.1.3 GLN formulation

Given a neuron j at the i-th layer, its output is obtained from Eq. (3.1):

$$h^{(i)}_{j,(\mathbf{x},\mathbf{z})} = \sigma\left(\left(\mathbf{w}^{(i)}_{j,(\mathbf{z})}\right)^\top \sigma^{-1}\left(\mathbf{h}^{(i-1)}_{(\mathbf{x},\mathbf{z})}\right)\right),\ i \geq 1 \tag{3.7}$$

with $\mathbf{h}^{(0)}_{(\mathbf{x},\mathbf{z})} = \sigma(\mathbf{x})$ and $\mathbf{w}^{(i)}_{j,(\mathbf{z})} \in \mathbb{R}^{d_{i-1}}$ the weight vector associated with the region activated by the context $\mathbf{z}$ for the corresponding neuron according to Eq. (3.6).

Finally, rewriting Eq. (3.7) in matrix form akin to Eq. (3.2), the output for the i-th layer (with d_i neurons) for the input-context pair $(\mathbf{x}, \mathbf{z})$ is

$$\mathbf{h}^{(i)}_{(\mathbf{x},\mathbf{z})} = \sigma\left(\mathbf{W}^{(i)}_{(\mathbf{z})} \sigma^{-1}\left(\mathbf{h}^{(i-1)}_{(\mathbf{x},\mathbf{z})}\right)\right),\ i \geq 1 \tag{3.8}$$

with $\mathbf{h}^{(0)}_{(\mathbf{x},\mathbf{z})} = \sigma(\mathbf{x})$. $\mathbf{W}^{(i)}_{(\mathbf{z})} \in \mathbb{R}^{d_i \times d_{i-1}}$ is a matrix where each row is the weight vector associated to the region activated by the context $\mathbf{z}$ for the corresponding neuron.

3.1.4 Training procedure of GLNs

GLNs have been introduced to operate in online and continual learning settings, mainly because of the local training nature and sample efficiency.

Training the intermediate neurons of a GLN does not rely on backpropagating the errors through the layers. Instead, each neuron is trained as a standalone classifier to predict the (binary) target [8].

We result in each neuron being trained independently optimizing a convex problem. As you can see from the example reported in Figure 3.1, the binary target is simultaneously propagated to all the neurons of the network. Since

each neuron predicts the output, we can directly compute the gradient based on the neuron prediction, without propagating the error downwards. This allows to update all neurons' weights vector in a single step.

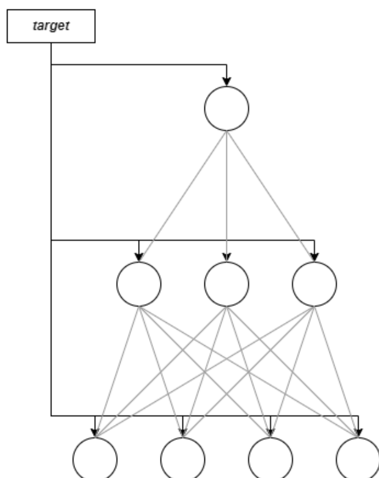


Figure 3.1: Training procedure of GLN. The target is directly propagated to all the neurons, which compute a local gradient to update the weights.

Moreover, the gating mechanism in each neuron selects a subset of the input data for each region, that are used (in an online setting) to train the weights of the geometric mixer associated to that region. Finding the weights of a geometric mixer is a convex problem that is much easier to solve (i.e. requires fewer data samples and fewer iterations) with respect to the non-linear optimization usually associated to neural networks. In particular, the original paper shows GLNs performance to be competitive with MLPs or SVMs with just a single pass over the training data. GLNs achieve non-linearity via a gating mechanism (random halfspace gating) that is not trained and different for each node. This mechanism allows to decouple the errors of each neuron from the ones of the others, thus obtaining an effective layer-wise representation.

3.2 GLN analysis

In GLNs, an effective gating mechanism should partition the context space to optimize the neuron utilization and make the downstream task easier to solve. In practice, this may translate into finding a partition of the space such that the data in each region is well-fitted by a model of the form of (3.1). By this argument, it follows that the choice of the hyperplanes for the gating should be principled and reflect –at least in part– the data distribution. Otherwise, the context space split would produce empty-regions, which are useless, because the associated weights would never be selected and trained.

For instance, consider the regions produced by a single neuron in the example reported in Figure 3.2. The black data points are divided into seven separate groups by the red hyperplanes, because each hyperplane cuts the populated part of the space in two. Hence, the dataset is used to train seven different sets of weights. Instead, the blue hyperplanes are badly initialized and creates only one non-empty region, thus only the weights associated to the region full of example are trained, and the neuron is equivalent to a linear neuron.

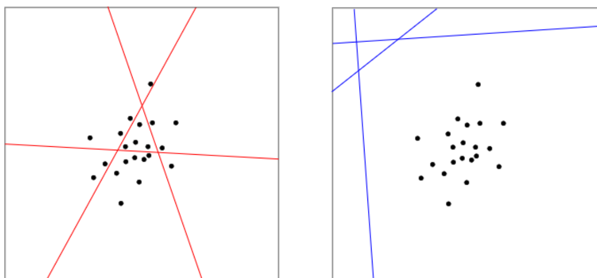


Figure 3.2: Example of different region split. The same 2D dataset (black dots) are split into seven different regions by the red hyperplanes (left), while the blue hyperplanes are not able to separate the data, generating only a single non-empty region (right).

In GLNs, the hyperplanes used to performed halfspace gating are randomly sampled from a normal distribution fixed beforehand. In particular, given a hyperplane $(\mathbf{a}, b)$, $\mathbf{a} \in \mathbb{R}^{d^{(z)}}$, $b \in \mathbb{R}$, we have

$$\mathbf{a} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

$$b \sim \mathcal{N}(0, \sigma^2)$$

where $\mathbf{0} \in \mathbb{R}^{d^{(z)}}$ is the zero vector, $\mathbf{I}$ is the identity matrix in $\mathbb{R}^{d^{(z)} \times d^{(z)}}$ and σ^2 a small fixed value.

This choice of hyperplanes generation it is easy to implement and it has the nice property of producing orthogonal hyperplanes with high probability. However, it also introduces a strong bias on the data distribution, because fixing σ^2 to a small value implies small values for b, hence the hyperplanes pass very close to the origin. This implies that data should be approximately centered in the origin for a hyperplane to split them into two groups.

For this reason, before feeding data into GLNs, they are initially standardized. The standardization is performed feature-wise, i.e. each component of the context vector $\mathbf{z}$ is transformed according to

$$z_i \leftarrow \frac{z_i - \mu_i}{\sigma_i}$$

A variation used by [46] in their code is by dividing by $\sigma_i + 1$. However, because the true expected value μ_i and standard deviation σ_i are not available, the authors used the Welford's online algorithm [48] to incrementally estimate them

in an online fashion, by recursively updating the current estimates according to the following formula

$$\mu_i(t+1) = \mu_i(t) + \frac{z_i(t+1) - \mu_i(t)}{t+1}$$

$$\sigma_i^2(t+1) = \frac{m_i(t+1)}{t+1}$$

where

$$m_i(t+1) = m_i(t) + (x_i(t+1) - \mu_i(t))(x_i(t+1) - \mu_i(t+1)),$$

$\mu_i(1) = z_i$ and $m_i(1) = 0$.

The above iterations provide more and more accurate estimates of $\boldsymbol{\mu} := [\dots, \mu_i, \dots]^\top, \boldsymbol{\sigma} := [\dots, \sigma_i, \dots]^\top$ as time progresses, however it is associated with two potential setbacks. The estimates improvement is guaranteed only when the sequence of context vectors $\mathbf{z}$ is stationary and i.i.d., not always a realistic assumption in continual learning tasks. Secondly, when performing predictions on one of the initially learned tasks, it is possible that the GLN parameters learned during the early stages have been trained with respect to a data standardization that is substantially different from the current one. If so, it is unlikely that the performance will remain comparably good when encountering the same task in the future.

To show that random halfspace gating introduces a strong bias on the data distribution and that Welford's algorithm is not well suited for continual learning, we perform experiments considering different approaches to standardize the context features and calibrate the space partitioning with the data distribution.

As a first baseline method, we consider an offline standardization, where the standardization parameter vectors $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$ are computed from all the task together, as if we knew also the future tasks. Of course, this approach is not applicable in practice, nevertheless, it serves as a gold standard to discern the different factors impacting on the training. We refer to this method as *feature-wise offline* standardization, as opposed to the Welford's data standardization, which is feature-wise but online, and the following baseline method that is not feature-wise.

Another possible approach consists in using a simpler standardization scheme, in which we consider a single mean and standard deviation $\mu, \sigma \in \mathbb{R}$ over all context feature components. The context vectors are then standardized (centered and scaled) as explained above. Also in this case, the statistics are computed offline to avoid any effect caused by online estimation, and call the method *global* (offline) standardization. When the distribution of context features is overall the same across tasks, an accurate online estimation of the standardization parameters should be obtained quicker by this method, because of the fewer number of unknown parameters (only 2, μ and σ, instead of $d^{(z)}$). In turn, this makes the online version comparable to the offline one.

Finally, as last baseline, we consider applying no data standardization. This last method serves as a lower bound on the performance.

The experimental results are reported and analyzed in Section 5.2.

Chapter 4

Prototypes Gated Linear Networks

As discussed in Chapter 3, the initialization of the hyperplanes of GLNs can be an issue in a continual learning setting. A solution would be to initialize the hyperplanes in a data-driven fashion, however, this is not straightforward. We would need to first estimate the region of space occupied by the data and then choose a set of hyperplanes that pass through that region.

In this chapter, we propose an alternative GLN gating mechanism that is based on prototypes instead of random hyperplanes. Relying on prototypes makes it easier to exploit the knowledge about where regions should be distributed in the gating mechanism. In fact, since prototypes live in the context space, for example, we could simply select some relevant data samples as prototypes and they will induce a region split that follows the data distribution.

Moreover, using prototypes instead of hyperplanes to define the region has an advantage in terms of the possible splittings that can be generated.

We refer to the GLN with such prototype-based gating mechanism as Prototype Gated Linear Network (ProtoGLN).

In Section 4.3, we present a prototype initialization method that solves ProtoGLN's initialization issues and make it applicable to practical continual learning scenarios.

The model, called Growing Prototype Gated Linear Network (GrowPGLN), is able to adapt to changes in the data distribution expanding the number of prototypes when needed. The expansion of prototypes is controlled by a removal mechanism, which removes useless prototypes according to a simple heuristic.

4.1 Prototype-based gating mechanism

ProtoGLNs extend GLNs defining a new context function used by the gating mechanism. Thus, the formulation of ProtoGLN is the same of Eq. 3.8

The context function c presented in Section 3.1.2 and exploited in (3.6) is based on random halfspace gating.

We propose an alternative approach, which consist in defining a partition of the space based on a set of prototypes. Each point in the space is then assigned to the closest prototype, obtaining a Voronoi tessellation. Figure 4.1 compares the region split obtain with both prototypes and random hyperplanes.

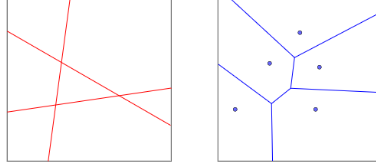


Figure 4.1: Region split in 2D produced by of halfspace gating (left) and gating with prototypes (right). The split produced by prototypes corresponds to a Voronoi Tessellation.

To define this new gating mechanism, let us consider a matrix $\mathbf{P}_j^{(i)} \in \mathbb{R}^{k \times d^{(z)}}$ of prototypes associated to the j-th neuron of layer i. The gating vector $\mathbf{c}_{j,(\mathbf{z})}^{(i)} \in \{0,1\}^k$ can be formulated as:

$$\mathbf{c}_{j,(\mathbf{z})}^{(i)} = \text{one_hot}\left(\arg\min_l(\|\mathbf{p}_{j,l}^{(i)} - \mathbf{z}\|)\right) \tag{4.1}$$

where $\mathbf{p}_{j,l}^{(i)}$ is the l-th row of $\mathbf{P}_j^{(i)}$ and $\|\cdot\|$ is the 2-norm assessing the distance between each row $\mathbf{p}_{j,l}^{(i)}$ and context vector $\mathbf{z}$.

4.2 Prototypes initialization

Depending on the prototypes initialization, ProtoGLNs can suffer from the same issues of GLNs. In fact, a naïve random initialization strategy, presents the same problems of randomly initialized hyperplanes. For instance, we could initialize each prototype $\mathbf{p} \in \mathbb{R}^{d^{(z)}}$ sampling it from a multivariate normal distribution, like

$$\mathbf{p} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$$

where $\mathbf{0}$ is the zero vector in $\mathbb{R}^{d^{(z)}}$ and $\mathbf{\Sigma}$ is a diagonal covariance matrix with small-value variances.

With this assumption, we introduce the same bias on the data distribution of GLNs, thus the data have to be standardized, as discussed in Section 3.2.

However, this issue could be avoided applying a data-driven initialization strategy. Though it is quite complex to compute the hyperplanes starting from the data, it is trivial with prototypes, because they live in the context space.

In a traditional machine learning scenario, an easy way to initialize prototypes could be to just sample them uniformly from the whole training set $Tr = \bigcup_{i=1}^{T} Tr^{(i)}$, i.e. $\mathbf{p}_{j,l}^{(i)} \sim \mathcal{U}(Tr)$, where $\mathbf{p}_{j,l}^{(i)}$ is the l-th row of $\mathbf{P}_j^{(i)}$ and $\mathcal{U}(S)$

denotes the uniform distribution over the elements of a finite set S. Of course this is not directly applicable to a continual learning setting and Section 4.3 explores a more realistic data-driven initialization approach. However, it can be used as a first upper-bound on the models performance, where we assume to have knowledge of the entire dataset in advance.

The potential of ProtoGLN, compared to GLNs, is shown by the experimental results in Chapter 5.

4.3 Growing ProtoGLN

In Section 4.2, we discussed how the prototype initialization method is crucial for the model to learn continually.

A good initialization strategy should consider two fundamental aspects:

1. prototypes should be close enough to the data manifold, such that each region is non-empty
2. the data distribution may change over time, thus prototypes should be continually updated

For these reasons, we define a mechanism that incrementally add new prototypes to the model, selecting them from new data according to a distance-based policy. Doing so, the prototypes exactly falls on the data manifold and they adapt according to the changing data distribution. We call this growing version of the model GrowPGLN. The complete growth mechanism is presented in Sec. 4.3.1.

The idea of adapting a set of prototypes to the data distribution is not new in literature and it is at the base of models like Neural Gas (NG) [29] and Self-Organizing Maps (SOM) [20]. They basically consist in estimating the data distribution by iteratively updating the prototypes, moving them towards seen examples. However, both SOM and NG presents a fix number of adaptable prototypes.

The first method that introduced a growth mechanism is Growing Neural Gas (GNG) [11], which extends the Neural Gas model, letting new prototypes to be allocated when needed, depending on the complexity of the data distribution.

This mechanism has been improved by Incremental Growing Neural Gas (IGNG) [37], which increase the convergence speed of Growing Neural Gas by allocating new prototypes when a sample falls far from the current set of prototypes.

Our growth mechanism is inspired by the Neural Gas family of models and exploits a distance-based policy, similar to IGNG, but with a significant difference: we stick to the idea of allocating new prototypes when a sample is very far from current prototypes, however, we do not move the prototypes once allocated.

In fact, moving the prototypes would change a neuron's region split, causing the learned weights vectors to be used on different part of the space. Hence, the weights are no more related to the region of space they were trained on.

Note that it is not necessary to move prototypes to approximate the data distribution, because they are chosen from the set of samples.

4.3.1 Growth mechanism

Let us consider a single neuron for simplicity, because the same mechanism is applied in parallel to all neurons before each training step.

The *Grow* function, defined by the pseudocode of Algorithm 1, add a prototype only if the set of prototypes is empty or if the distance of the current context vector from the closest prototype is greater than a threshold d. In this way new prototypes (and new regions) are added only where the context space has not been explored yet.

The value of d is strictly related to the average distance between the samples of dataset and has to be set carefully. A small value of d would lead to a huge number of prototypes, while, on the other hand, with a high value of d only few prototypes are added. Figure 4.2 intuitively shows the importance of the hyperparameter d. Starting from an initial prototype (red point), if the value of d is too small (left), any sample outside the red circle can be added to the set of prototypes. When d is properly set (center), only the distant points can be added. Finally, when d is too high (right), none of the samples can become a prototype.

If we assume that the average distance between the samples of different tasks is the same, then we could compute the value on a subset $\mathcal{S}$ of examples coming from the first task as

$$d = \frac{\sum_{1 \leq i < j \leq |\mathcal{S}|} ||\mathbf{x}_i - \mathbf{x}_j||}{\frac{|\mathcal{S}| \cdot (|\mathcal{S}| - 1)}{2}} \tag{4.2}$$

where $\mathbf{x}_i, \mathbf{x}_j \in \mathcal{S}$.

Otherwise, the hyperparameter has to be chosen with standard hyperparameter search techniques, e.g. performing a grid search over a range of values.

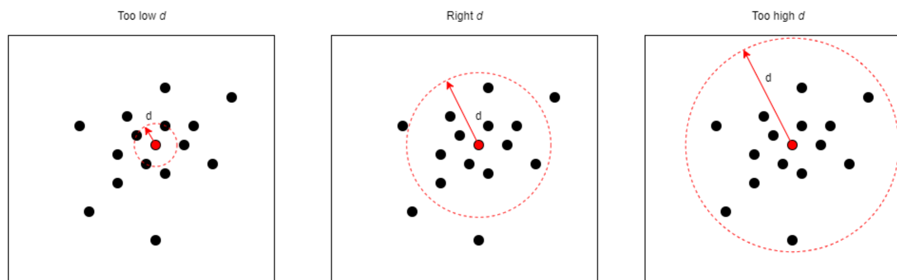


Figure 4.2: Example of the importance of the distance threshold d. When the value of d is too low (left), almost any new sample becomes a prototype, while with a high value of d (right) no new prototype is added. Only with the right value of d (center), a fair number of prototypes can be added.

Algorithm 1: Grow($\mathbf{z}$,$\mathcal{P}$,t)

```
Input: context vector z
Input: set of prototypes P
Input: threshold d
Output: updated set of prototypes P
if P = ∅ then
  | P := {z};
else
  | dist := ∞;
  | foreach p ∈ P do
  |   | dist ← min(dist, ||p − z||)
  | end
  | if dist ≥ d then
  |   | add z to P
  | end
end
return P
```

To make the effect of the growing mechanism more clear, consider the case of a new sample fed to the model. If its position in the context space is close to one of the prototypes (which are past data), it means that the sample follows the same distribution of previous data and the local space has already been partitioned.

On the other hand, when the sample is far from all the prototypes, we can have two possibilities: either the data distribution is shifting to a new location, or the sample is an outlier.

Finding a change in the data distribution is the goal of the growing procedure, thus adding a new prototype and a new region in the former case is the right behavior. Instead, when the sample is an outlier caused by noise, it means that in its surroundings there are no other samples and adding a new region would be pretty useless. However, even though it is useless, adding outliers as prototypes cause no harm to the model, because when a new prototype is added, the Voronoi tessellation is changed locally, without affecting far regions. Moreover, outliers can be safely removed by the removal mechanism discussed in Section 4.3.3.

Dropout for neuron diversification

For each neuron, the growth mechanism is deterministic, thus the same data stream will produce the same context space partition. This is a problem for GrowPGLNs, because their strength is based on the different selection of the neurons weights depending on the context vector.

If all neurons present the same set prototypes, the final region split would be identical. A direct effect of an identical region split is that different examples falling in the same regions would select the same set of weights. This means

that having a network where each neuron has k prototypes is equivalent to have k linear models, one for each region of space, completely nullifying the non-linearity introduced by the gating mechanism.

To differentiate the neurons, we decide to apply the well studied dropout mechanism [44]. Since neurons are randomly deactivated, they see only a random subset of the samples. Hence, they are trained on different data streams, leading to different sets of prototypes and final region splits.

Dropout basically consist in deactivating a random subset of the neurons of the network at each training step, and it was originally thought to improve the generalization ability of neural networks and prevent them from overfitting the training set. In fact, dropout is a form of bagging, an ensemble technique which consist in training different models on subsets of the data. In this case, the different models are the subnetworks obtained dropping part of the neurons.

Usually ensemble methods require aggregating the predictions of all the models of the ensemble, meaning that the storage and computational complexity at prediction time is linear in the number of models. With neural networks this can be avoided applying the weight scaling rule, as proved in [3], which consist in multiplying the output of each neuron times $1 - p_{drop}$, with p_{drop} dropout probability of the neuron.

To prove that dropout can be applied to GrowPGLNs (and in general to GLNs), we start from the proof of the weight scaling rule for Deep Linear Networks presented in [3]. The proof is based on a probabilistic approach and for ease of comprehension it has been adapted using the matrix notation.

If we consider a single linear layer (without activation), the output can be expressed in matrix form as

$$\mathbf{o} = \mathbf{W}\mathbf{x}$$

with $\mathbf{x} \in \mathbb{R}^n$ input and $\mathbf{W} \in \mathbb{R}^{m \times n}$ weights matrix.

We can apply dropout to the layer simply defining a binary mask $\mathbf{d} \sim \text{Bernoulli}(\mathbf{q})$, where $\mathbf{q} = [q_1, ..., q_n]$ is a vector of probabilities obtained from the dropout probability vector $\mathbf{p}_{drop}$ as $\mathbf{q} = \mathbf{1} - \mathbf{p}_{drop}$. The mask $\mathbf{d}$ corresponds to a multivariate Bernoulli random vector.
The mask is then multiplied element-wise to the input, obtaining

$$\mathbf{o} = \mathbf{W}(\mathbf{d} \odot \mathbf{x}) \tag{4.3}$$

The output vector $\mathbf{o}$ is a multivariate random variable, since $\mathbf{d}$ is generated by a stochastic process. Thus, we can compute the expected value of $\mathbf{o}$ w.r.t. all possible masks $\mathbf{d} \sim \text{Bernoulli}(\mathbf{q})$.

For ease of notation we denote $\mathbb{E}_{\mathbf{d} \sim \text{Bernoulli}(\mathbf{q})}$ just with $\mathbb{E}$.

The expected value of $\mathbf{o}$ can be computed as

$$\begin{aligned}\mathbb{E}[\mathbf{o}] &= \mathbb{E}[\mathbf{W}(\mathbf{d} \odot \mathbf{x})] \\ &= \mathbf{W}(\mathbb{E}[\mathbf{d} \odot \mathbf{x}]) \\ &= \mathbf{W}(\mathbb{E}[\mathbf{d}] \odot \mathbf{x}) \\ &= \mathbf{W}(\mathbf{q} \odot \mathbf{x})\end{aligned} \tag{4.4}$$

because $\mathbf{W}$ and $\mathbf{x}$ do not depend on the stochastic process, thus they are treated like constants and can be taken out of the expectation. Note that $\mathbb{E}[\mathbf{d}] = \mathbf{q}$, because the expected value of a Bernoulli random variable is equal to the probability of it being equal to one.

The same reasoning can easily be extended to a multi-layer network. Let us consider a deep linear network with L layers. The output of each layer can be computed with the following formula

$$\mathbf{h}^{(i)} = \mathbf{W}^{(i)}\mathbf{h}^{(i-1)}, \; i \geq 1$$

where $\mathbf{h}^{(0)} = \mathbf{x}$.
Then, applying the binary dropout mask $\mathbf{d}^{(i)} \sim \text{Bernoulli}(\mathbf{q}^{(i)})$ similarly to Eq. (4.3) we obtain

$$\mathbf{h}^{(i)} = \mathbf{W}^{(i)}(\mathbf{d}^{(i)} \odot \mathbf{h}^{(i-1)}), \; i \geq 1 \tag{4.5}$$

Therefore, the expected value of the output of each layer w.r.t. all possible dropout masks is

$$\mathbb{E}[\mathbf{h}^{(i)}] = \mathbb{E}[\mathbf{W}^{(i)}(\mathbf{d}^{(i)} \odot \mathbf{h}^{(i-1)})] \tag{4.6}$$

$$= \mathbf{W}^{(i)}\mathbb{E}[\mathbf{d}^{(i)} \odot \mathbf{h}^{(i-1)}] \tag{4.7}$$

$$= \mathbf{W}^{(i)}(\mathbb{E}[\mathbf{d}^{(i)}] \odot \mathbb{E}[\mathbf{h}^{(i-1)}]) \tag{4.8}$$

$$= \mathbf{W}^{(i)}(\mathbf{q}^{(i)} \odot \mathbb{E}[\mathbf{h}^{(i-1)}]) \tag{4.9}$$

Differently from before, in this case $\mathbf{h}^{(i-1)}$ depends on the stochastic process, so the expected value of the output depends on the expected value of the input.

Unrolling the recursive formula we obtain that the expected value of the output of the final layer is

$$\begin{aligned}\mathbf{o} = \mathbf{h}^{(L)} &= \mathbf{W}^{(L)}(\mathbf{q}^{(L)} \odot \mathbb{E}[\mathbf{h}^{(L-1)}]) \\ &= \mathbf{W}^{(L)}(\mathbf{q}^{(L)} \odot \mathbf{W}^{(L-1)}(\mathbf{q}^{(L-1)} \odot \cdots \mathbf{W}^{(1)}(\mathbf{q}^{(1)} \odot \mathbf{x})))\end{aligned} \tag{4.10}$$

If we define $\mathbf{Q}^{(i)} = \text{diag}(\mathbf{q}^{(i)}) \; \forall i \in [1, ..., L]$, Eq. (4.10) can be rewritten as

$$\mathbf{o} = \mathbf{W}^{(L)}\mathbf{Q}^{(L)} \cdots \mathbf{W}^{(1)}\mathbf{Q}^{(1)}\mathbf{x} \tag{4.11}$$

We can see that, assuming the dropout probability to be the same for all the neurons of a layer, we have

$$\mathbf{Q}^{(i)} = q_i\mathbf{I}$$

with $\mathbf{I}$ identity matrix of same shape of $\mathbf{Q}^{(i)}$.
Finally, substituting this in Eq. (4.11), we obtain

$$\begin{aligned} \mathbf{o} &= \mathbf{W}^{(L)} q_L \mathbf{I} \cdots \mathbf{W}^{(1)} q_1 \mathbf{I} \mathbf{x} \\ &= \mathbf{W}^{(L)} q_L \cdots \mathbf{W}^{(1)} q_1 \mathbf{x} \end{aligned} \tag{4.12}$$

which corresponds to multiply each weights matrix times $1 - p_{drop}$ of the corresponding layer or, equivalently, to multiply the output of each layer times $1 - p_{drop}$.

Therefore, the weight scaling rule is exact for deep linear networks.

It is possible to show that the rule is exact also if we assign a different dropout probability to each neuron, but the proof would require additional considerations and this assumption is sufficient for our purpose.

We proved that applying the weight scaling rule to a deep linear network for the output computation is equivalent to compute the expected value of the output w.r.t. all possible dropout masks, thus it is equivalent to bagging.

Showing that this results holds also for GrowPGLNs is straightforward making some considerations.

First, GLN-based models are quite different from a simple linear network. In fact, the weights selection using a context function introduces a non-linearity.

However, after we fix the input of the network that will be used as side information, the weights are selected and fixed as well. So we can separate the two phases (weight selection and output computation) and assume that the selection of the weights is done before the output computation.

Another aspect is the application of the logit and sigmoid function to the input and output of each layer respectively. However, this is not a problem because their cumulative effect is null.

Therefore, after the weight selection step, we have a standard deep linear network, as shown in Eq. 3.5 (the final sigmoid activation can be ignored). Thus, the weight scaling rule is exact also for GLN-based models.

4.3.2 GrowPGLN Formulation

As for Prototypes GLNs, the final network formulation is the same of Eq. 3.8. However, the growing nature of prototypes introduces a difference on the context function.

In ProtoGLNs, as discussed in Section 4.1, the number of rows of the prototypes matrix $\mathbf{P}_j^{(i)} \in \mathbb{R}^{k \times d^{(z)}}$ is fixed to k, the hyperparameter corresponding to the number of regions, and the gating vector $\mathbf{c}_{j,(\mathbf{z})}^{(i)}$ is defined in $\{0,1\}^k$.

Instead, in GrowPGLN, the number of regions is variable and can also be different from neuron to neuron thanks to the differentiation introduced by dropout. Thus, the number of regions k associated to a neuron becomes a function of time, obtaining $\mathbf{P}_j^{(i)} \in \mathbb{R}^{k_{ij}(t) \times d^{(z)}}$ and $\mathbf{c}_{j,(\mathbf{z})}^{(i)} \in \{0,1\}^{k_{ij}(t)}$. Instead, the context function formulation is unchanged and it is the same of Eq. 4.1.

4.3.3 Removal mechanism

The growth mechanism described in Section 4.3.1 has an evident drawback: there is not a limit to the number of prototypes added by each neuron. The absence of a fixed limit makes sense in the continual learning framework, however for practical application the memory requirement might become an issue. This drawback is strongly related to the insertion policy, which is sensible to outliers and cause an unnecessary additional partition of a neuron's regions, as argued in 4.3.1.

For this reason, we decide to add a simple removal mechanism, based on the amount of training samples falling in each region. The idea is straightforward: for each prototype we define a counter which stores the number of samples assigned to it. Then, the removal policy just consist in periodically removing the regions whose counter is less or equal than a given threshold, as defined by the pseudocode of Algorithm2. The *Removal* function is defined over a single neuron and is then applied to all neurons in parallel.

The mechanism is very simple but it is perfect to remove outlier prototypes. To do so, we fix the threshold to 1, because empty regions are trained only on a single sample, which is to the related prototype. Note that the time of application of the removal mechanism is important, because also non-outlier prototypes can result as trained on a single sample as soon as they are inserted.

The removal of a region, and the associated weights vector, can potentially destroy the representation learned by the model, as shown in Figure 4.3. In the example, a neuron initially have five prototypes, generating five regions. Assume to remove the central prototypes: all the samples falling into the red shaded area, would now be assigned to the closest of the remaining prototypes. However, the remaining prototypes are not trained on the reassigned sample, thus, for those samples, the neuron could predict completely wrong results.

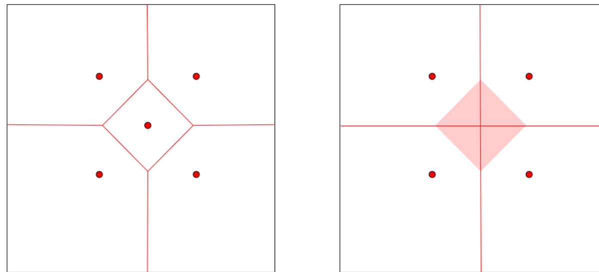


Figure 4.3: Example of a neuron's prototype before (left) and after (right) the removal of the prototype in the middle. When a prototype is removed, the corresponding region is split and incorporated in the neighbors regions. After the removal, samples falling into the red shaded area are assigned to the closest of the remaining prototypes.

However, if a region is empty, the related weights vector is not trained, thus the corresponding prototype can be safely removed.

The results reported in Section 5.4.2 show that applying this policy we can

significantly improve the memory consumption of GrowPGLN, without reducing the performance of the model.

Algorithm 2: Removal($\mathcal{P}$,n)

Input: set of prototypes $\mathcal{P}$
Input: threshold n
Output: updated set of prototypes $\mathcal{P}$
foreach $\mathbf{p} \in \mathcal{P}$ **do**
  if *samples(*$\mathbf{p}$) $\leq n$ **then**
    remove $\mathbf{p}$ from $\mathcal{P}$
  end
end
return $\mathcal{P}$

Chapter 5

Experiments

In this chapter, we show and analyze the results obtained by the discussed models on two continual learning benchmarks: Permuted MNIST and Split MNIST.

The analysis highlights the limitations of GLNs, in particular the standardization issue. Moreover, we show that our method, based on an alternative data-driven gating mechanism, is robust to data standardization and it shows almost no forgetting on the chosen benchmarks.

5.1 Experimental setting

This section reports the experimental setting to ease the reproducibility of the experiments.

5.1.1 Models

We implement and analyze three different models: GLN, ProtoGLN and GrowPGLN. The models are developed using Python and they are based on the PyTorch[1] library. We make the code for replicating the experiments available online[2]. For all the models, the weights of each neuron are learned incrementally using online gradient descent; see Section 3.1.4 for further details.

Before running the experiments concerning the original GLN model, we had care to verify the reproducibility of the results from the original paper.

5.1.2 Hyperparameters

To ease the results analysis, we decide to fix the hyperparameters shared by the models. In this way it is simpler to understand what is the cause of an improvement or decay in the performance.

[1]https://pytorch.org/
[2]https://github.com/matteo-munari/GrowPGLN

Two hyperparameters are fixed for all the experiments and models, using the values reported in the original paper [46]:

- Model architecture: we keep the same number of layers and neurons defined in the original GLN paper. Thus, all the models are based on a 3-layer GLN with $100 \rightarrow 25 \rightarrow 1$ Gated linear neurons;
- Learning rate: we fix the learning rate to 0.01.

Other hyperparameters (e.g. number of regions, model initialization, dropout rate) might vary depending on the dataset or the model. Hence, they are reported in each experiment section.

Finally, for what concerns the context vectors $\mathbf{z}$, in practical applications (and in the original GLN paper), the input and the context vector are the same. In this paper, we consider this case as well.

5.1.3 Benchmarks

Usual Machine Learning datasets are not well suited for training Continual Learning models, because they generally deal with a single task. Such features are too restrictive for a Continual Learning setting, that is way completely new dataset have been developed.

For historical reasons, Continual Learning datasets are mainly related to the vision and reinforcement learning fields, where retraining a model from scratch requires a lot of time.

Most of continual learning datasets derives from standard machine learning dataset. A common method consists in applying some transformation to the dataset examples, one for each task. Another approach consist in splitting a dataset into parts and associating each sub-dataset to a different task.

Table 5.1 reports some examples of Continual Learning benchmarks.

Table 5.1: Continual Learning Benchmarks

Dataset	**Tasks**	**Samples per task**
Permuted MNIST [13]	8	60 000
Split MNIST [49]	5	$\sim$ 10 000
Split CIFAR-10 [49]	5	10 000
Split CIFAR-100 [49]	10 / 20	5000 / 2500
MNIST-SVHN [18]	2	$\sim$ 60 000
Omniglot [21]	10-50	$\sim$ 20

In this work, we focus on the Permuted MNIST and Split MNIST benchmarks.

Permuted MNIST [13] is a variation of MNIST where a random permutation is applied to the pixels of the images, without changing the image labels. The task consists in classifying the ten MNIST digits, but with the pixels permuted

using the task-associated permutation. Thus, Permuted MNIST is well suited for continual learning, because we can define an arbitrarily long sequence of tasks by random sampling different permutations. In our work, we create a sequence of 8 tasks. An important feature of this dataset is that the different tasks are rarely conflicting against each other, because the number of possible permutations is huge compared to the number of considered tasks and input data from different task result almost orthogonal.

Figure 5.1 shows an example of Permuted MNIST data. A random permutation is applied to the original MNIST digits (upper part) obtaining completely different images. The permuted images are meaningless for humans, but the pixels still follow a pattern, since similar digits lead to similar permuted digits. The images are not processed as 28x28 matrix, but as a unique vector in $\mathbb{R}^{784}$.

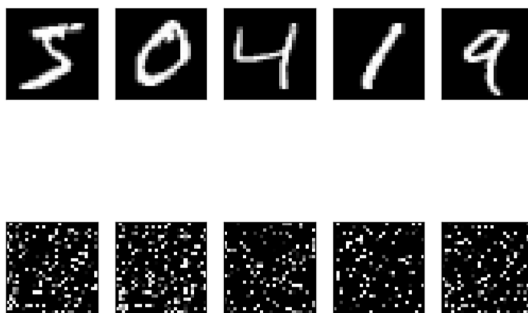


Figure 5.1: Example of a single task samples of Permuted MNIST. Each MNIST digit (upper images) is transformed using a random pixel permutation, obtaining a different image (lower images). The permuted images are meaningless for humans, but a neural network should be able to find the characteristic pattern of a digit, independently of the pixels relative position.

The second dataset is Split MNIST [49], another this variation of MNIST. The original dataset is split into five tasks, with each task presenting a binary classification problem over two subsequent digits. The most common split, which we adhere to, consist in dividing the digits as follows: (0,1), (2,3), (4,5), (6,7), (8,9); note that each digit appears in only one of the 5 tasks.

The digit split is shown in Figure 5.2.

Also in this case, the images are flattened and represented as vectors in $\mathbb{R}^{784}$. Differently from Permuted MNIST, in this dataset relevant pixels are shared by different tasks, thus they might interfere with each other.

We discussed how in the Permuted MNIST tasks do not interfere with each other, while in the Split samples from different tasks are close to each other.

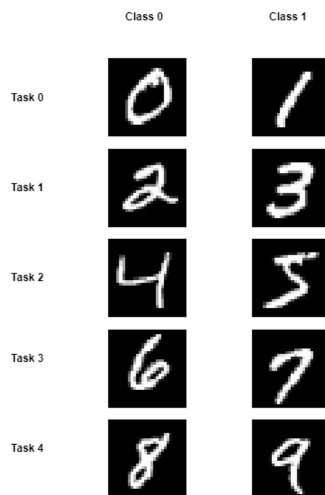


Figure 5.2: Task division in Split MNIST. Each task consist in a binary classification problem between two digits. The goal is to assign even digits to class 0 and odd digits to class 1.

To see this qualitatively, we could plot the data distribution, but since the data space is highly dimensional ($\mathbb{R}^{784}$) this is not straightforward.

We can do this using UMAP[3], an open-source tool that can be used to reduce the data dimensions to two, making them visualizable in 2D. An important feature of UMAP exploit data locality to perform the transformation, with the effect that close samples in the high dimension space are likely to be close also in their reduced representation. Note, however, that distances in the reduced space are not representative of the original space.

Figure 5.3 show the bidimensional representation of the Permuted MNIST training set distribution. We can easily identify 8 clusters divided by color, representing the eight different task. This means that the samples of each task falls in a different region of space.

Figure 5.4, instead, shows the Split MNIST data distribution. In this case, there is not a sharp separation between samples of different task, with many data points falling in the same region of samples of other tasks.

5.1.4 Evaluation

Defining a protocol to evaluate continual learning models is really challenging. Firstly, because the learning process might continue forever, thus we can not simply test the model performance at the end of training. Secondly, the environment conditions (training set and task) change over time, meaning that also the test samples should change over time.

The most popular evaluation protocol consist in defining for each task i a

[3] `https://github.com/lmcinnes/umap`

Figure 5.3: Permuted MNIST dataset 2D representation using UMAP. The samples are colored by task. Samples of the same task are similar to each other and there is a sharp distinction between different tasks.

training set $Tr^{(i)}$ and a test set $Te^{(i)}$. In this way, we can separately measure the performance of the model on each task.

During training, it is possible to compute the accuracy on all test sets. Usually, this is done exactly after learning each task, to compare the model performances before and after learning a given task. These accuracies are collected and visualized in a matrix $R \in \mathbb{R}^{N \times N}$, called accuracy matrix, with N number of tasks, where each entry $R_{i,j}$ represents the accuracy of the model in solving task j, after learning task i. An example of accuracy matrix is reported in Table 5.2. Accuracy matrix are useful to visualize the evolving of a model performance in time, reading the columns downwards starting from the top one.

Using the accuracy matrix, it is possible to compute different metrics, which highlight different aspect of the model performances. For example, they can give a measure of the overall accuracy of the model or of the amount of forgetting.

There are three important metrics that we can compute using the accuracy matrix: Average Accuracy, Backward Transfer and Forward Transfer [27].

The Average Accuracy (ACC), computed as

$$ACC = \frac{1}{N} \sum_{i=1}^{N} R^{(N,i)} \tag{5.1}$$

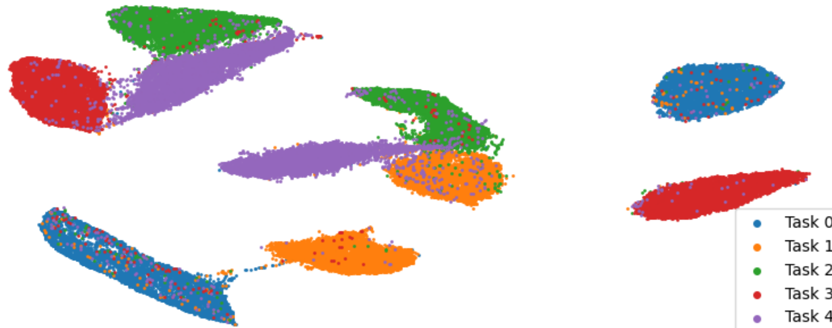


Figure 5.4: Split MNIST dataset 2D representation using UMAP. The samples are colored by task. There is no sharp distinction between samples of different tasks.

Table 5.2: Example of accuracy matrix

	$Te^{(1)}$	$Te^{(2)}$	$Te^{(3)}$
$Tr^{(1)}$	$R_{1,1}$	$R_{1,2}$	$R_{1,3}$
$Tr^{(2)}$	$R_{2,1}$	$R_{2,2}$	$R_{2,3}$
$Tr^{(3)}$	$R_{3,1}$	$R_{3,2}$	$R_{3,3}$

gives a measure of the ability of the model in solving all learned tasks at the end of training. It is useful when we are interested in the final performances of the model.

Backward Transfer (BWT), instead, shows how much the learning of later tasks increases or decreases the initial performances. The metric assumes a positive value if the model accuracy on old tasks increases as new tasks are seen, while it is negative if there is forgetting. BWT is computed as

$$BWT = \frac{1}{N-1} \sum_{i=1}^{N-1} R_{N,i} - R_{i,i} \tag{5.2}$$

Forward Transfer (FWT) is useful when we are interested in the effect that learning a task has on future tasks. A high value of FWT means that the model is able to reuse already learn knowledge to solve other tasks, performing transfer learning. FWT is computed as

$$BWT = \frac{1}{N-1} \sum_{i=2}^{N} R_{i-1,i} - \bar{b}_i \tag{5.3}$$

where $\bar{b}_i$ is the test accuracy on task i computed as soon as the model is randomly initialized.

To globally consider the performance of the model, and not only at the end of training, in [25] the author introduces a variation of the ACC metric, defining it as

$$A = \frac{\sum_{i \geq j}^{N} R_{i,j}}{\frac{N(N+1)}{2}} \tag{5.4}$$

5.2 Context standardization on GLNs

In this experiment we compare the performance –and in particular forgetting phenomena– obtained by combining the halfspace gating of Section 3.1.2 with the four standardization strategies previously discussed in Chapter 3, namely,

1. online feature-wise standardization (original implementation with Welford's method);
2. offline feature-wise standardization (gold standard);
3. offline global standardization (lower bound);
4. no standardization (second lower bound).

For the Permuted MNIST dataset, the number of hyperplanes of the model is set to 6, generating 2^6 regions, while on the Split MNIST dataset this hyperparameter is increased to 12, obtaining 2^{12} regions. All the following results are computed as mean over five runs.

We report the results achieved on the Permuted MNIST dataset in Figure 5.5.

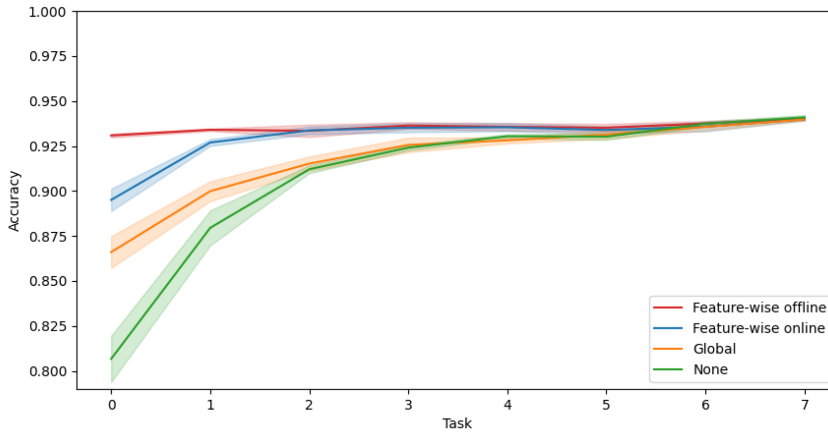


Figure 5.5: Comparison between different methods for data standardization on Permuted MNIST using GLNs. Shaded areas correspond to +/- one standard deviation.

The blue line shows the results using Welford's online algorithm for standardization, that corresponds to the results reported in the original GLN paper.

It is possible to see that the model with this standardization performs poorly on the first (and, to a lesser extent, on the second) task, w.r.t. the others. Table 5.3, reports the accuracy matrix relative to the model, as defined in Section 5.1.4. The empty cells, representing the accuracies on future tasks, are not relevant for the study of forgetting, hence they are not computed.

From the first column of the table, we can see that initially the model is able to achieve high accuracy on the first task (first row), however, the more new tasks are seen, the more the accuracy drops. This behavior is not present in the other tasks.

Interestingly, we found that this anomalous forgetting is strictly related to the data standardization. In fact, when considering the *gold standard* offline feature-wise standardization (red line), we see that the model does not show any sign of forgetting, i.e., its performance is stable on all previously learned tasks. The good performance is due to both the offline nature of this standardization (i.e. the transformation is fixed throughout learning) and the good feature centering effect, that places the input data in a good region for the considered hyperplanes. The global offline standardization (orange line) shows lower performance, probably because it is not able to properly center the features (more on this aspect later). Finally, using no standardization at all results in the lowest performance, since the features are not centered.

The behavior of global and no standardization methods can be explained analyzing the behavior of the halfspace gating mechanism of GLNs. The gating hyperplanes are randomly sampled in a way such that they are orthogonal to each other with high probability. Since the bias is sampled from a normal distribution with small mean, the points of intersection of multiple hyperplanes are close to the origin; thus, the data distribution should be approximately centered on the origin for the model to split the data at the maximal extent.

Table 5.3: Accuracy matrix (%) on Permuted MNIST with GLNs (feature-wise online standardization).

Training	**Test Set**							
Set	$\mathbf{Te^{(0)}}$	$\mathbf{Te^{(1)}}$	$\mathbf{Te^{(2)}}$	$\mathbf{Te^{(3)}}$	$\mathbf{Te^{(4)}}$	$\mathbf{Te^{(5)}}$	$\mathbf{Te^{(6)}}$	$\mathbf{Te^{(7)}}$
$\mathbf{Tr^{(0)}}$	94.5 ± 0.1	-	-	-	-	-	-	-
$\mathbf{Tr^{(1)}}$	93.6 ± 0.1	94.2 ± 0.2	-	-	-	-	-	-
$\mathbf{Tr^{(2)}}$	93.1 ± 0.2	93.9 ± 0.1	94.2 ± 0.1	-	-	-	-	-
$\mathbf{Tr^{(3)}}$	92.4 ± 0.2	93.7 ± 0.2	93.8 ± 0.2	94.1 ± 0.1	-	-	-	-
$\mathbf{Tr^{(4)}}$	91.9 ± 0.1	93.5 ± 0.1	93.8 ± 0.2	93.8 ± 0.2	94.1 ± 0.2	-	-	-
$\mathbf{Tr^{(5)}}$	91.1 ± 0.3	93.1 ± 0.1	93.6 ± 0.1	93.7 ± 0.2	93.8 ± 0.2	94.0 ± 0.2	-	-
$\mathbf{Tr^{(6)}}$	90.3 ± 0.4	93.1 ± 0.2	93.5 ± 0.2	93.6 ± 0.2	93.7 ± 0.1	93.4 ± 0.2	94.0 ± 0.2	-
$\mathbf{Tr^{(7)}}$	89.5 ± 0.6	92.7 ± 0.2	93.4 ± 0.2	93.5 ± 0.2	93.5 ± 0.2	93.4 ± 0.2	93.6 ± 0.3	94.0 ± 0.1

In Figure 5.6, instead, we report the results of the different standardizations applied to GLN on the Split MNIST dataset. In this case, the task that shows the higher forgetting is the third one, probably because the learning of the fourth or fifth task interferes with the learned representation. The observations done previously hold also in this case. In fact, using offline feature-wise standardization gives quite stable performance in all tasks, while with the online standardization the accuracies are a bit lower. Moreover, with global or no standardization, there is a significant drop in performance on the first tasks.

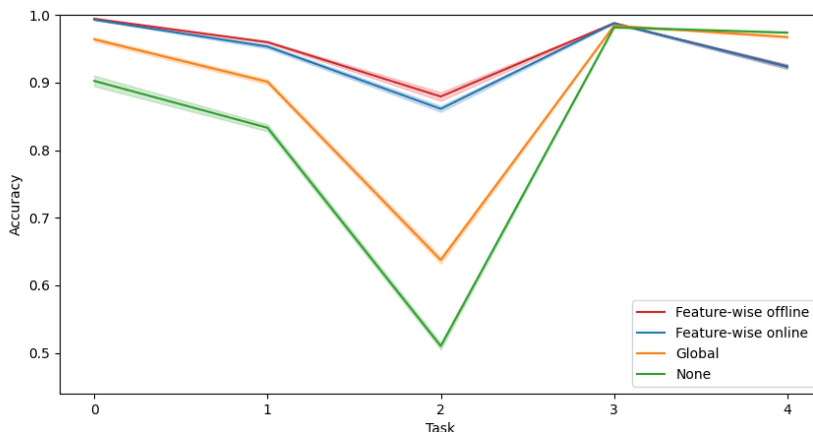


Figure 5.6: Comparison between different methods for data standardization on Split MNIST using GLNs. Shaded areas correspond to +/- one standard deviation.

However, even the best model (GLN with feature-wise offline standardization), presents a quite high amount of forgetting on the third task, as we can see from Table 5.4. The performances on the third task (third column), drop significantly after learning the last task. This is probably due to the closeness of the samples of the third and last tasks, as discussed in Section 5.1.3, meaning that the last task samples selects and update the same set of weights trained for the third task.

In general, from the plots we can observe that two factors influence the model performance:

1. the less centered the data is, the more forgetting the model exhibits;
2. fixing the statistics dimensionality, the online standardization policy tends to show more forgetting than the offline one.

5.3 Prototypes GLN

With this experiment, we show that substituting halfspace gating with gating based on prototypes, presented in Chapter 4, leads to better results. To make

Table 5.4: Accuracy matrix (%) on Split MNIST with GLNs (feature-wise offline standardization).

Training Set	**Test Set**				
	$\mathbf{Te^{(0)}}$	$\mathbf{Te^{(1)}}$	$\mathbf{Te^{(2)}}$	$\mathbf{Te^{(3)}}$	$\mathbf{Te^{(4)}}$
$\mathbf{Tr^{(0)}}$	99.8 ± 0.1	-	-	-	-
$\mathbf{Tr^{(1)}}$	99.6 ± 0.0	98.1 ± 0.1	-	-	-
$\mathbf{Tr^{(2)}}$	99.6 ± 0.0	98.0 ± 0.2	98.2 ± 0.2	-	-
$\mathbf{Tr^{(3)}}$	99.5 ± 0.1	97.1 ± 0.1	96.6 ± 0.3	98.9 ± 0.1	-
$\mathbf{Tr^{(4)}}$	99.3 ± 0.0	96.5 ± 0.0	87.5 ± 0.5	99.1 ± 0.1	95.4 ± 0.4

these results comparable with the ones obtained by the experiment in Section 5.2, we set the number of prototypes to $2^6 = 64$ on the Permuted MNIST dataset and to $2^{12} = 4096$ on Split MNIST. In this way the number of regions per neuron is the same between models on the same dataset.

The experiment is performed with two initialization policies: a data-driven initialization, which consist in sampling the prototypes from the whole dataset, and a random initialization, where prototypes are sampled from a normal distribution.

5.3.1 Data-driven prototypes initialization

Initializing prototypes in a data-driven way, we found that not only the model is more robust to the choice of statistics used for data standardization, but also it is capable of achieving better accuracy.

Figures 5.7 and 5.9 (magnified in Figures 5.8 and)show how a model with prototypes sampled uniformly from the entire dataset reaches high performance on the Permuted MNIST and Split MNIST datasets, respectively, independently of the data standardization. Notice that the only data standardization method that shows some noticeable forgetting in the first task of Permuted MNIST is Welford's online algorithm. As discussed for the halfspace gating case, the model is initially able to achieve an high accuracy on the first task, but as training continues this accuracy drops, as shown by the first column of Table 5.5. Even in this case, this forgetting is most likely due to the artificial concept drift introduced by the online estimation of the standardization parameters, because it is not present in successive task and with other type of standardization.

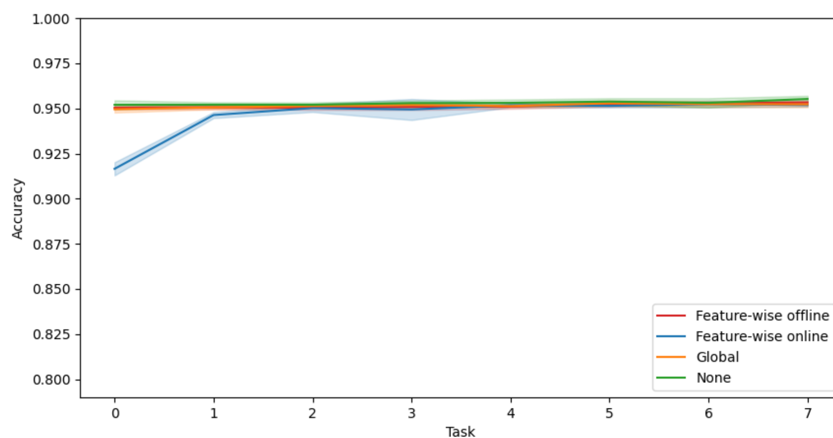


Figure 5.7: Comparison between different methods for data standardization on Permuted MNIST using ProtoGLN (data-driven initialization). Shaded areas correspond to +/- one standard deviation.

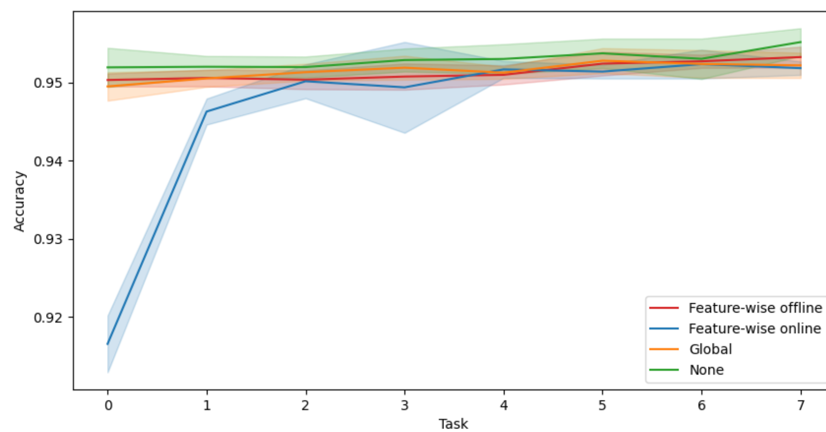


Figure 5.8: Comparison between different methods for data standardization on Permuted MNIST using ProtoGLN (data-driven initialization). Shaded areas correspond to +/- one standard deviation. (magnification)

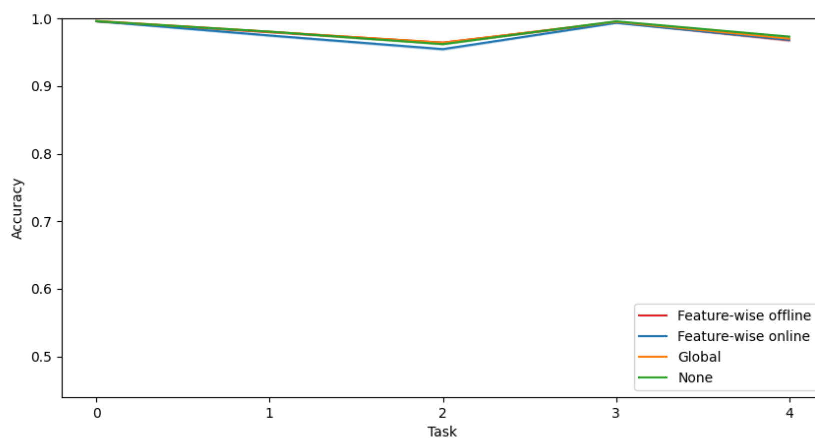


Figure 5.9: Comparison between different methods for data standardization on Split MNIST using ProtoGLN (data-driven initialization). Shaded areas correspond to +/- one standard deviation.

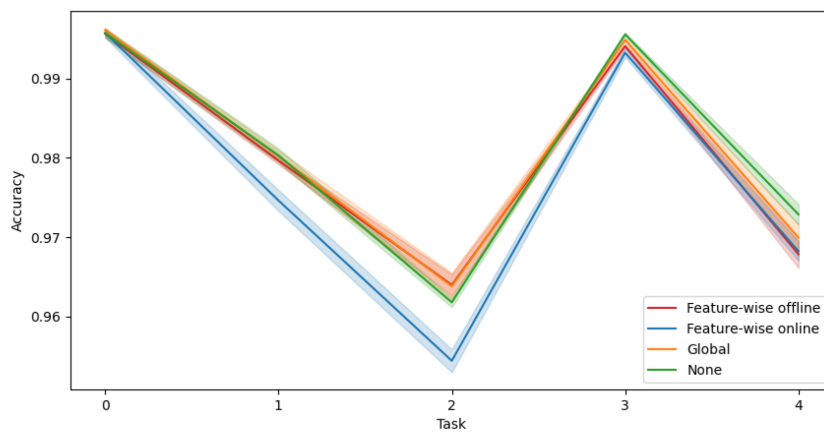


Figure 5.10: Comparison between different methods for data standardization on Split MNIST using ProtoGLN (data-driven initialization). Shaded areas correspond to +/- one standard deviation. (magnification)

Table 5.5: Accuracy matrix (%) on Permuted MNIST with data-driven ProtoGLN (feature-wise online standardization).

Training Set	**Test Set**							
	$\mathbf{Te}^{(0)}$	$\mathbf{Te}^{(1)}$	$\mathbf{Te}^{(2)}$	$\mathbf{Te}^{(3)}$	$\mathbf{Te}^{(4)}$	$\mathbf{Te}^{(5)}$	$\mathbf{Te}^{(6)}$	$\mathbf{Te}^{(7)}$
$\mathbf{Tr}^{(0)}$	95.2 ± 0.1	-	-	-	-	-	-	-
$\mathbf{Tr}^{(1)}$	94.5 ± 0.2	95.1 ± 0.2	-	-	-	-	-	-
$\mathbf{Tr}^{(2)}$	93.7 ± 0.1	95.0 ± 0.1	95.0 ± 0.1	-	-	-	-	-
$\mathbf{Tr}^{(3)}$	93.0 ± 0.2	94.9 ± 0.1	95.2 ± 0.1	95.0 ± 0.5	-	-	-	-
$\mathbf{Tr}^{(4)}$	92.7 ± 0.3	94.9 ± 0.1	95.1 ± 0.1	94.9 ± 0.6	95.2 ± 0.1	-	-	-
$\mathbf{Tr}^{(5)}$	92.2 ± 0.4	94.8 ± 0.1	95.2 ± 0.1	94.9 ± 0.5	95.2 ± 0.1	95.2 ± 0.1	-	-
$\mathbf{Tr}^{(6)}$	91.8 ± 0.4	94.7 ± 0.2	95.1 ± 0.1	94.9 ± 0.4	95.2 ± 0.1	95.2 ± 0.1	95.3 ± 0.1	-
$\mathbf{Tr}^{(7)}$	91.7 ± 0.4	94.6 ± 0.2	95.0 ± 0.2	94.9 ± 0.6	95.2 ± 0.1	95.1 ± 0.1	95.2 ± 0.2	95.2 ± 0.1

5.3.2 Random prototypes initialization

In Figure 5.11 we report the results on Permuted MNIST of the random prototypes-based strategy with different standardization methods. While the relative order of the different standardizations is the same compared to the random halfspace gating, comparing with Figure 5.5, we note that, fixing the data standardization method, random prototypes always show less forgetting compared to random halfspace gating. The same observations can be made on the results on Split MNIST reported in Figure 5.12.

However, if we compare random prototypes and data-driven prototypes, we can see that the latter's performances are higher on both dataset, even in the ideal case where feature-wise offline standardization is applied.

This shows that a data-driven prototypes initialization is more effective that simply drawing them randomly.

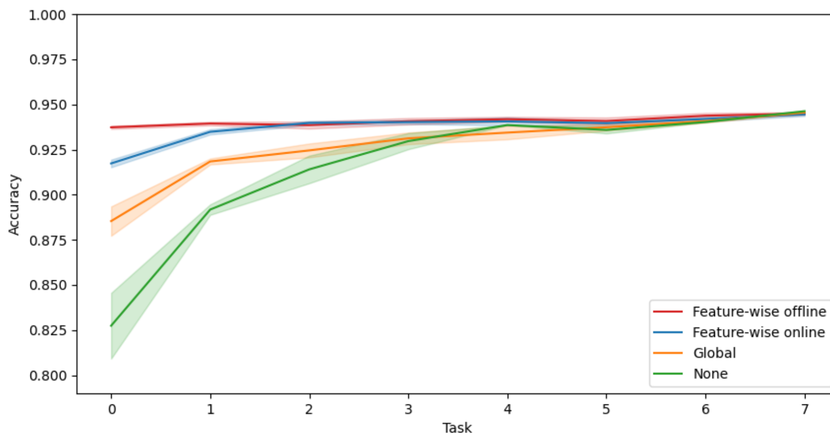


Figure 5.11: Comparison between different methods for data standardization on Permuted MNIST using ProtoGLN (random initialization). Shaded areas correspond to +/- one standard deviation.

5.4 GrowPGLN

In this section we report the results regarding the experiments on the growing version of ProtoGLN and we show that GrowPGLN is a practical data-driven approach for continual learning.

We perform two distinct experiments, which study the model with and without the removal mechanism, respectively in Section 5.4.1 and Section 5.4.2. GrowPGLN presents a different set of hyperparameters with respect to the other models. In particular, the dropout rate p_{drop} and the distance threshold d have to be set.

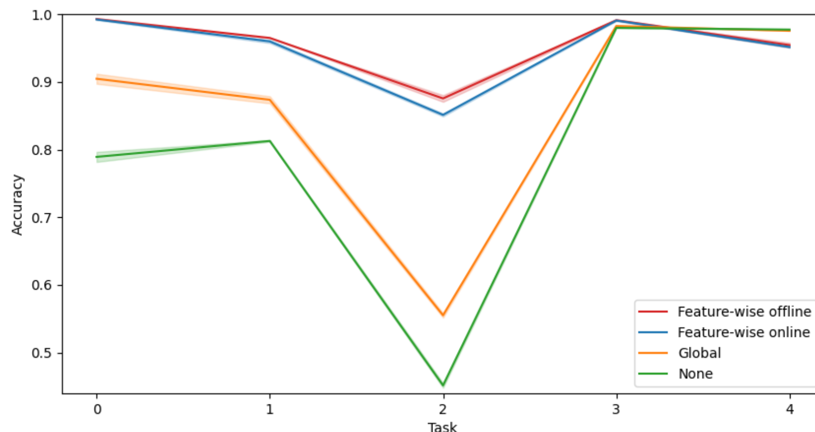


Figure 5.12: Comparison between different methods for data standardization on Split MNIST using ProtoGLN (random initialization). Shaded areas correspond to +/- one standard deviation.

The dropout rate is selected applying a grid search over a set of possible values. The distance threshold is set to $d = 10.16$ on Permuted MNIST and it is computed as the average distance between samples of the first task, according to Eq. (4.2), because this value is similar for all the tasks. On Split MNIST, instead, this assumption does not hold, thus the value for d is selected using grid search. The results on Split MNIST are relative to $d = 6.0$.

Finally, since GrowPGLN is robust to different data standardizations, we choose to consider for both dataset the original data, without applying any standardization.

5.4.1 Results with growth mechanism only

Figure 5.13 shows the final accuracy obtained by GrowPGLN on the Permuted MNIST dataset. We can observe that lines representing the accuracy of GrowPGLN with the different dropout value, fall between the upper bound of data-driven ProtoGLN (green line) and the lower bound of random ProtoGLN (full red line). Note that the lower bound refers to the model with offline feature-wise data standardization. With no data standardization, the performance of ProtoGLN are the ones showed by the red dotted line. However, GrowPGLN performances are better the random ProtoGLN even if we consider the optimal case where we can apply the feature-wise standardization.

The same pattern can be observed also on Split MNIST as shown by Figure 5.14, except for the results of the model with dropout probability = 0.3 (purple line), which presents a significant forgetting on task 0.

The plots suggest that, with an optimal dropout value, the growing approach

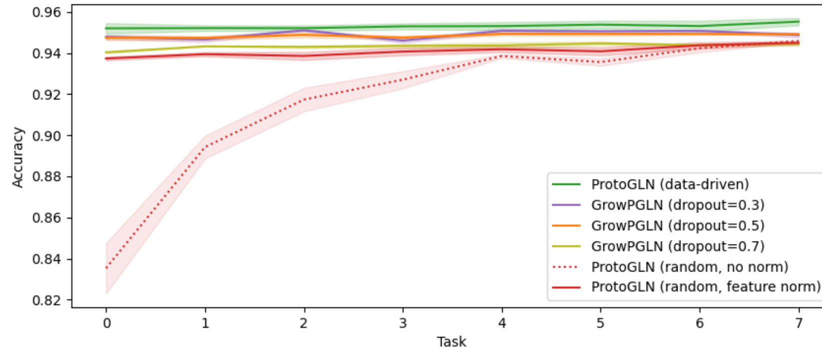


Figure 5.13: Comparison between ProtoGLN and GrowPGLN with different dropout values on Permuted MNIST. All the results obtained with GrowPGLN are between the upper bound (green) of data-driven ProtoGLN and the lower bounds (red full and dotted lines) of random ProtoGLN. Shaded areas correspond to +/- one standard deviation.

is able to compete with data-driven ProtoGLN, almost reaching the upper bound without knowing the whole training set at the beginning of training. Moreover, note that GrowPGLN presents almost no forgetting, on both Permuted and Split MNIST, as Table 5.6 and 5.7 show. The results refer to a dropout probability of 0.5 on Permuted MNIST and of 0.7 on Split MNIST. The only reduction in performance occurs on the third task of Split MNIST, however note that the amount of forgetting is smaller than that of random ProtoGLN reported in Tab. 5.4.

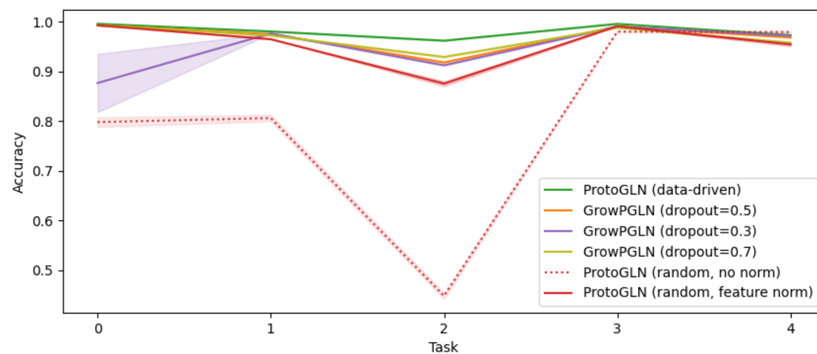


Figure 5.14: Comparison between ProtoGLN and GrowPGLN with different dropout values on Split MNIST. The results obtained by GrowPGLN with dropout values set to 0.5 and 0.7 are between the upper bound (green) of data-driven ProtoGLN and the lower bounds (red full and dotted lines) of random ProtoGLN. Instead, GrowPGLN with dropout set to 0.3 presents a drop on the first task. Shaded areas correspond to +/- one standard deviation.

Table 5.6: Accuracy matrix (%) on Permuted MNIST with GrowPGLN (dropout rate = 0.5).

Training Set	**Test Set**							
	$\mathbf{Te^{(0)}}$	$\mathbf{Te^{(1)}}$	$\mathbf{Te^{(2)}}$	$\mathbf{Te^{(3)}}$	$\mathbf{Te^{(4)}}$	$\mathbf{Te^{(5)}}$	$\mathbf{Te^{(6)}}$	$\mathbf{Te^{(7)}}$
$\mathbf{Tr^{(0)}}$	94.9 ± 0.1	-	-	-	-	-	-	-
$\mathbf{Tr^{(1)}}$	94.8 ± 0.1	94.8 ± 0.0	-	-	-	-	-	-
$\mathbf{Tr^{(2)}}$	94.8 ± 0.1	94.8 ± 0.0	94.9 ± 0.1	-	-	-	-	-
$\mathbf{Tr^{(3)}}$	94.8 ± 0.1	94.8 ± 0.1	94.9 ± 0.1	94.8 ± 0.1	-	-	-	-
$\mathbf{Tr^{(4)}}$	94.8 ± 0.1	94.8 ± 0.1	94.9 ± 0.1	94.8 ± 0.1	94.9 ± 0.1	-	-	-
$\mathbf{Tr^{(5)}}$	94.7 ± 0.1	94.8 ± 0.1	94.9 ± 0.1	94.8 ± 0.1	94.9 ± 0.1	94.9 ± 0.1	-	-
$\mathbf{Tr^{(6)}}$	94.7 ± 0.1	94.7 ± 0.1	94.9 ± 0.1	94.8 ± 0.1	94.9 ± 0.1	94.9 ± 0.1	94.9 ± 0.1	-
$\mathbf{Tr^{(7)}}$	94.7 ± 0.1	94.7 ± 0.1	94.9 ± 0.1	94.7 ± 0.1	94.9 ± 0.1	94.9 ± 0.1	94.9 ± 0.0	94.9 ± 0.0

Table 5.7: Accuracy matrix (%) on Split MNIST with GrowPGLN (dropout rate = 0.7).

Training Set	**Test Set**				
	$\mathbf{Te^{(0)}}$	$\mathbf{Te^{(1)}}$	$\mathbf{Te^{(2)}}$	$\mathbf{Te^{(3)}}$	$\mathbf{Te^{(4)}}$
$\mathbf{Tr^{(0)}}$	100.0 ± 0.0	-	-	-	-
$\mathbf{Tr^{(1)}}$	99.8 ± 0.0	99.1 ± 0.0	-	-	-
$\mathbf{Tr^{(2)}}$	99.5 ± 0.0	99.0 ± 0.0	99.5 ± 0.0	-	-
$\mathbf{Tr^{(3)}}$	99.3 ± 0.1	98.1 ± 0.1	98.0 ± 0.1	98.7 ± 0.0	-
$\mathbf{Tr^{(4)}}$	99.3 ± 0.0	97.2 ± 0.1	92.9 ± 0.1	98.8 ± 0.0	95.8 ± 0.1

GrowPGLNs can be applied to practical continual learning scenarios, without the additional assumptions of GLNs and ProtoGLNs.

However, the choice of the hyperparameters, in particular the dropout rate, is crucial to avoid forgetting. For example, analyzing the accuracy matrix obtained with GrowPGLN with a dropout probability of 0.3 (Table 5.8) and 0.5 (Table 5.9), we can see that after learning the third task (third row), the accuracy on the first task presents a sharp drop. The model, however, is able to recover learning the fourth task. This effect is probably due to the similarities between samples of different tasks of Split MNIST, as discussed in Section 5.1.3. Note that with a higher level of dropout, Table 5.7, this behavior does not present. The results suggests that when the tasks interfere with each other, a higher dropout rate is preferable.

Table 5.8: Accuracy matrix (%) on Split MNIST with GrowPGLN (dropout rate = 0.3).

Training Set	**Test Set**				
	$\mathbf{Te^{(0)}}$	$\mathbf{Te^{(1)}}$	$\mathbf{Te^{(2)}}$	$\mathbf{Te^{(3)}}$	$\mathbf{Te^{(4)}}$
$\mathbf{Tr^{(0)}}$	100.0 ± 0.0	-	-	-	-
$\mathbf{Tr^{(1)}}$	99.3 ± 0.5	99.2 ± 0.1	-	-	-
$\mathbf{Tr^{(2)}}$	74.4 ± 2.6	99.0 ± 0.0	99.8 ± 0.0	-	-
$\mathbf{Tr^{(3)}}$	94.0 ± 3.5	98.4 ± 0.0	98.4 ± 0.0	98.9 ± 0.0	-
$\mathbf{Tr^{(4)}}$	87.7 ± 5.8	97.6 ± 0.0	91.2 ± 0.1	99.0 ± 0.1	97.2 ± 0.2

5.4.2 Results with both growth and removal mechanisms

To show the effectiveness of the removal mechanism we re-run the experiment presented in the previous section, with the additional prototypes removal step. We compare only the growing models which gave the best results in the previous experiment, thus we set the dropout probability to 0.5 on Permuted MNIST and to 0.7 on Split MNIST. To better study the removal mechanism, we apply it at the end of each task, to avoid any deletion of newly inserted prototypes.

Figure 5.15 and Figure 5.16 show the results obtained by GrowPGLN with

Table 5.9: Accuracy matrix (%) on Split MNIST with GrowPGLN (dropout rate = 0.5).

Training Set	**Test Set**				
	$\mathbf{Te}^{(0)}$	$\mathbf{Te}^{(1)}$	$\mathbf{Te}^{(2)}$	$\mathbf{Te}^{(3)}$	$\mathbf{Te}^{(4)}$
$\mathbf{Tr}^{(0)}$	100.0 ± 0.0	-	-	-	-
$\mathbf{Tr}^{(1)}$	99.8 ± 0.0	99.1 ± 0.1	-	-	-
$\mathbf{Tr}^{(2)}$	94.0 ± 5.3	99.0 ± 0.1	99.7 ± 0.0	-	-
$\mathbf{Tr}^{(3)}$	99.3 ± 0.0	98.4 ± 0.1	98.5 ± 0.1	98.9 ± 0.1	-
$\mathbf{Tr}^{(4)}$	99.3 ± 0.1	97.5 ± 0.0	91.8 ± 0.2	99.0 ± 0.1	96.8 ± 0.1

prototypes removal, respectively on Permuted and Split MNIST. The plot follow the same scheme of Figures 5.13 and 5.14. We immediately see that the performance are similar: for both dataset, the model accuracies fall between the upper and lower bound.

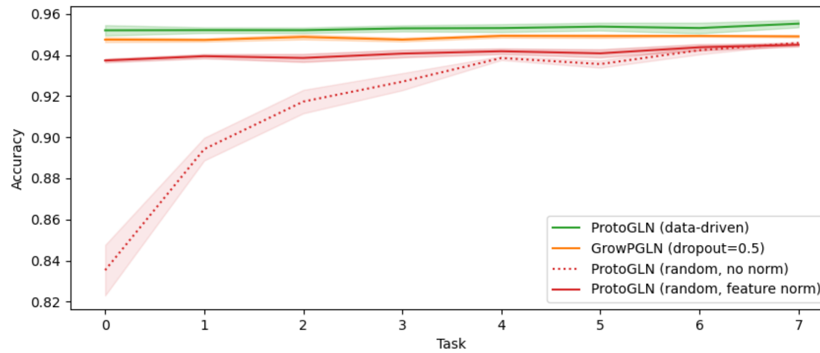


Figure 5.15: Comparison between ProtoGLN and GrowPGLN with removal mechanism on Permuted MNIST. The dropout rate of GrowPGLN is fixed to 0.5. The results obtained with GrowPGLN are between the upper bound (green) of data-driven ProtoGLN and the lower bounds (red full and dotted lines) of random ProtoGLN. Shaded areas correspond to +/- one standard deviation.

Figure 5.17 and Figure 5.18, instead, shows a direct comparison of the best performance of GrowGLN with (orange) and without removal mechanism (blue).

From the plots, it is evident that the removal mechanism almost does not affect the performance of the models. On Permuted MNIST, these results are not surprising, because analyzing the number of samples assigned to each region we found that only few regions (<1%) were empty and thus removed.

However, on the Split MNIST dataset the number of empty regions is very high, and more than half ($\sim$ 55%) of useless prototypes are removed.

This means that the simple removal policy, described in Sec. 4.3.3, can be an effective way to free up memory.

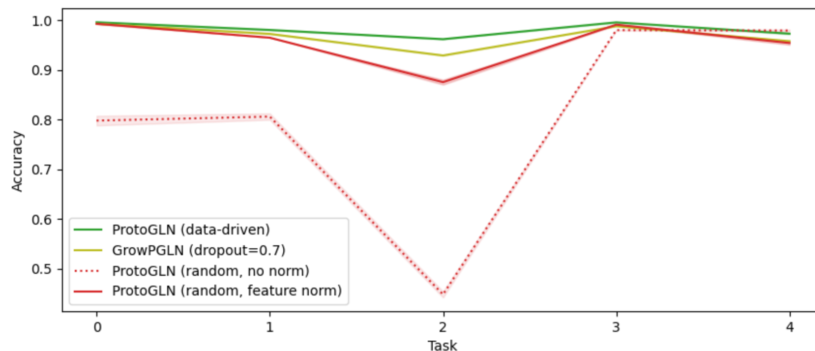


Figure 5.16: Comparison between ProtoGLN and GrowPGLN with removal mechanism on Split MNIST. The dropout rate of GrowPGLN is fixed to 0.7. The results obtained with GrowPGLN are between the upper bound (green) of data-driven ProtoGLN and the lower bounds (red full and dotted lines) of random ProtoGLN. Shaded areas correspond to +/- one standard deviation.

5.5 Models comparison

We can now use the results reported in the previous section to analyze and compare the different methods discussed in this work.

To make a quantitative comparison, we compute the average accuracy (A), defined by Eq. (5.4), for each model and dataset. We choose to use the metric A, instead of ACC of Eq. (5.1), to have a global measure of the learning process, not only a final snapshot. For the GrowPGLN models we consider only the model with dropout probability equal to 0.5 for the Permuted MNIST dataset and to 0.7 on Split MNIST. The measures are reported in Table 5.10.

The results are split into two groups: the upper part of the table reports the results relative to the methods that can be applied to a continual learning scenario, without assumptions on the data distribution. Instead, in the lower part of the table, we report the models that are not suited for real continual learning scenarios, because they assume to know some statistics on the whole dataset at the beginning of training. Thus, they are not good for learning from an infinite stream of data.

As we can see, GrowPGLN is the best model on both dataset, with an average accuracy at the end of training of about 94.8% on Permuted MNIST and 98.3% on Split MNIST. Note that these results are achieved with no standardization applied to the training data. Moreover, the performance of GrowPGLN are not so far from the upper bound of ProtoGLN (data-driven).

Table 5.11 ad 5.12, instead, compares the accuracies of the best three models that can be trained in a continual learning environment during the whole training process on the two datasets. The considered models are:

- GLN with online data standardization;

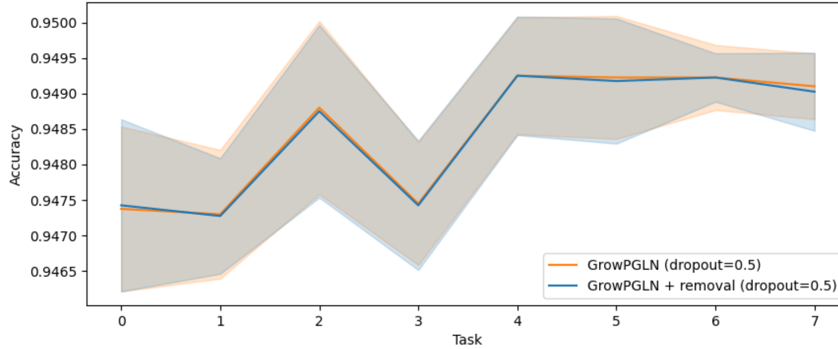


Figure 5.17: Comparison between GrowPGLN with (orange) and without (blue) removal mechanism on Permuted MNIST. The dropout rate of the GrowPGLN models is fixed to 0.5. The results obtained are almost identical. Shaded areas correspond to +/- one standard deviation.

- ProtoGLN with randomly initialized prototypes and online data standardization;
- GrowPGLN with no data standardization;

Each cell reports the accuracy obtained by GLN (top), ProtoGLN (middle) and GrowPGLN (bottom).

The results on Permuted MNIST show that GrowPGLN reaches the highest performance in every moment of the training process.

Also on Split MNIST, GrowPGLN achieves better performances than the other models, with the only exception on the accuracies on task 3 and on task 4 soon after learning them. However, the most important result is the accuracy of GrowPGLN on task 2 at the end of training, which is over 7% higher w.r.t. the other models.

Finally, in general, GrowPGLN presents almost no sign of forgetting, as discussed in Section 5.4.1.

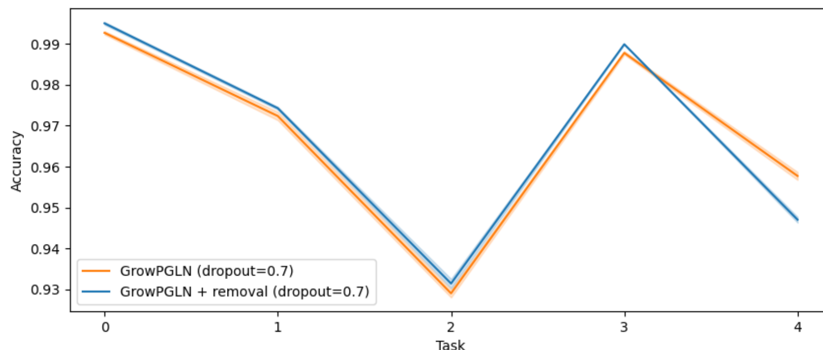


Figure 5.18: Comparison between GrowPGLN with (orange) and without (blue) removal mechanism on Split MNIST. The dropout rate of the GrowPGLN models is fixed to 0.7. The results obtained are almost identical, with a slight difference in the accuracy on the last task. Shaded areas correspond to +/- one standard deviation.

Table 5.10: Average percentage accuracy (A) of all tested models. The upper part of the table reports the results relative to the methods that can be directly applied to a continual learning scenario. The lower part reports the models that are not suited for real continual learning scenarios, but that are useful for the comparison. The results show that GrowPGLN achieve the best average accuracy on both datasets between the practical solutions. The model performances are also not so far from the upper bound of ProtoGLN.

Method	Standard.	Permuted MNIST (%)	Split MNIST (%)
GLN	online	93.3±1.1	96.3±3.7
GLN	none	91.5±3.5	92.1±12.3
ProtoGLN (random)	online	93.9±0.7	97.1±3.7
ProtoGLN (random)	none	92.2±2.9	90.0±14.5
GrowPGLN	**none**	**94.8±0.1**	**98.3±1.9**
GrowPGLN + removal	none	94.8±0.1	98.2±1.9
GLN	offline	93.7±0.3	96.7±3.3
GLN	global	92.6±1.6	94.7±9.0
ProtoGLN (data-driven)	online	94.6±1.0 98.3	±1.3
ProtoGLN (data-driven)	offline	95.2±0.1	98.6±1.0
ProtoGLN (data-driven)	global	95.2±0.1	98.7±1.0
ProtoGLN (data-driven)	**none**	**95.4±0.1**	**98.7±1.0**
ProtoGLN (random)	offline	94.2±0.3	97.5±3.1
ProtoGLN (random)	global	93.3±1.4	93.7±11.0

Table 5.11: Combination of accuracy matrix on Permuted MNIST of the three best models directly applicable to continual learning scenarios: GLN with online standardization, ProtoGLN (Proto) with online standardization, GrowPGLN (Grow) with no standardization (dropout = 0.5). Each cell reports the accuracy obtained by GLN (top), ProtoGLN (middle) and GrowPGLN (bottom), while each column shows the variation of the models' accuracies throughout the training process. GrowPGLN achieves a better accuracy in every task at every step of training . The best results of each cell are highlighted in bold.

Tr. Set	**Model**	**Test Set**							
		$\mathbf{Te}^{(0)}$	$\mathbf{Te}^{(1)}$	$\mathbf{Te}^{(2)}$	$\mathbf{Te}^{(3)}$	$\mathbf{Te}^{(4)}$	$\mathbf{Te}^{(5)}$	$\mathbf{Te}^{(6)}$	$\mathbf{Te}^{(7)}$
$\mathbf{Tr}^{(0)}$	GLN	94.5 ± 0.1	-	-	-	-	-	-	-
	Proto	94.6 ± 0.1	-	-	-	-	-	-	-
	Grow	**94.9±0.1**	-	-	-	-	-	-	-
$\mathbf{Tr}^{(1)}$	GLN	93.6 ± 0.1	94.2 ± 0.2	-	-	-	-	-	-
	Proto	94.0 ± 0.1	94.5 ± 0.2	-	-	-	-	-	-
	Grow	**94.8±0.1**	**94.8±0.1**	-	-	-	-	-	-
$\mathbf{Tr}^{(2)}$	GLN	93.1 ± 0.2	93.9 ± 0.1	94.2 ± 0.1	-	-	-	-	-
	Proto	93.3 ± 0.1	94.4 ± 0.2	94.4 ± 0.2	-	-	-	-	-
	Grow	**94.8±0.1**	**94.8±0.1**	**94.9±0.1**	-	-	-	-	-
$\mathbf{Tr}^{(3)}$	GLN	92.4 ± 0.2	93.7 ± 0.2	93.8 ± 0.2	94.1 ± 0.1	-	-	-	-
	Proto	93.1 ± 0.2	94.1 ± 0.2	94.3 ± 0.2	94.3 ± 0.1	-	-	-	-
	Grow	**94.8±0.1**	**94.8±0.1**	**94.9±0.1**	**94.8±0.1**	-	-	-	-
$\mathbf{Tr}^{(4)}$	GLN	91.9 ± 0.1	93.5 ± 0.1	93.8 ± 0.2	93.8 ± 0.2	94.1 ± 0.2	-	-	-
	Proto	92.8 ± 0.3	94.0 ± 0.2	94.3 ± 0.1	94.0 ± 0.3	94.5 ± 0.1	-	-	-
	Grow	**94.8±0.1**	**94.8±0.1**	**94.9±0.1**	**94.8±0.1**	**94.9±0.1**	-	-	-
$\mathbf{Tr}^{(5)}$	GLN	91.1 ± 0.3	93.1 ± 0.1	93.6 ± 0.1	93.7 ± 0.2	93.8 ± 0.2	94.0 ± 0.2	-	-
	Proto	92.5 ± 0.3	93.7 ± 0.2	94.2 ± 0.1	94.1 ± 0.2	94.1 ± 0.3	94.4 ± 0.1	-	-
	Grow	**94.7±0.1**	**94.8±0.1**	**94.9±0.1**	**94.8±0.1**	**94.9±0.1**	**94.9±0.1**	-	-
$\mathbf{Tr}^{(6)}$	GLN	90.3 ± 0.4	93.1 ± 0.2	93.5 ± 0.2	93.6 ± 0.2	93.7 ± 0.1	93.4 ± 0.2	94.0 ± 0.2	-
	Proto	92.0 ± 0.2	93.7 ± 0.1	94.2 ± 0.1	94.2 ± 0.1	94.1 ± 0.1	94.1 ± 0.1	94.5 ± 0.1	-
	Grow	**94.7±0.1**	**94.7±0.1**	**94.9±0.1**	**94.8±0.1**	**94.9±0.1**	**94.9±0.1**	**94.9±0.1**	-
$\mathbf{Tr}^{(7)}$	GLN	89.5 ± 0.6	92.7 ± 0.2	93.4 ± 0.2	93.5 ± 0.2	93.5 ± 0.2	93.4 ± 0.2	93.6 ± 0.3	94.0 ± 0.1
	Proto	91.7 ± 0.2	93.5 ± 0.1	94.0 ± 0.1	94.0 ± 0.1	94.1 ± 0.2	94.0 ± 0.1	94.2 ± 0.1	94.4 ± 0.1
	Grow	**94.7±0.1**	**94.7±0.1**	**94.9±0.1**	**94.7±0.1**	**94.9±0.1**	**94.9±0.1**	**94.9±0.1**	**94.9±0.1**

Table 5.12: Combination of accuracy matrix on Split MNIST of the three best models directly applicable to continual learning scenarios: GLN with online standardization, ProtoGLN (Proto) with online standardization, GrowPGLN (Grow) with no standardization (dropout = 0.7). Each cell reports the accuracy obtained by GLN (top), ProtoGLN (middle) and GrowPGLN (bottom), while each column shows the variation of the models' accuracies throughout the training process. GrowPGLN achieves a better accuracy in every task at every step of training . The best results of each cell are highlighted in bold.

Tr. Set	**Model**	**Test Set**				
		$\mathbf{Te^{(0)}}$	$\mathbf{Te^{(1)}}$	$\mathbf{Te^{(2)}}$	$\mathbf{Te^{(3)}}$	$\mathbf{Te^{(4)}}$
$\mathbf{Tr^{(0)}}$	GLN	99.6 ± 0.1	-	-	-	-
	Proto	99.8 ± 0.1	-	-	-	-
	Grow	**99.9±0.1**	-	-	-	-
$\mathbf{Tr^{(1)}}$	GLN	**99.6±0.1**	95.1 ± 0.3	-	-	-
	Proto	**99.6±0.1**	97.4 ± 0.2	-	-	-
	Grow	**99.6±0.1**	**98.8±0.1**	-	-	-
$\mathbf{Tr^{(2)}}$	GLN	**99.6±0.1**	96.2 ± 0.3	96.3 ± 0.2	-	-
	Proto	99.5 ± 0.1	97.9 ± 0.1	97.9 ± 0.1	-	-
	Grow	**99.6±0.1**	**98.8±0.1**	**99.3±0.1**	-	-
$\mathbf{Tr^{(3)}}$	GLN	**99.4±0.1**	95.1 ± 0.3	94.0 ± 0.3	98.6 ± 0.1	-
	Proto	**99.4±0.1**	96.4 ± 0.1	95.7 ± 0.2	**98.9±0.1**	-
	Grow	**99.4±0.1**	**98.0±0.1**	**97.7±0.1**	98.7 ± 0.1	-
$\mathbf{Tr^{(4)}}$	GLN	99.3 ± 0.1	95.3 ± 0.3	86.1 ± 0.4	98.8 ± 0.1	92.3 ± 0.3
	Proto	99.2 ± 0.1	96.0 ± 0.2	85.1 ± 0.2	**99.0±0.1**	**95.1±0.1**
	Grow	**99.5±0.1**	**97.4±0.1**	**93.1±0.1**	**99.0±0.1**	94.7 ± 0.1

Chapter 6

Conclusions

In this work, we proposed a new continual learning model based on Gated Linear Networks. Our model shows no sign of catastrophic forgetting, hence, it is able to learn a sequence of tasks, without forgetting hoe to solve old tasks when a new task is learned.

We first collected empirical evidence that allowed us to understand where the catastrophic forgetting exhibited by Gated Linear networks comes from. We considered two commonly adopted continual learning datasets for our benchmarks: Permuted MNIST and Split MNIST. In particular, we uncovered the relationship between hyperplane initialization in the halfspace gating mechanism and the distribution of the data in input, showing that data standardization is crucial for such a gating approach. Moreover, we show that the online estimation of standardization parameters has a significant negative impact on the amount of forgetting the model exhibits. For Permuted MNIST, we show that almost all the forgetting shown by GLNs is due to such standardization.

We then proposed an alternative gating mechanism that can be defined in a data-driven way, unleashing the potential of GLNs. With a proper gating, GLNs show almost no forgetting both in Permuted MNIST and Split MNIST. Moreover, we showed that the performance of GLNs with prototype-based gating improves over GLNs with halfspace gating, even when the prototypes are initialized without exploiting any information about the data distribution.

We also introduced an adaptive data-driven method for the initialization of prototypes, which incrementally adapt the gating mechanism to the data distribution respecting the continual learning assumptions.

Finally, we showed that the GrowPGLN model performs better than the other, showing almost no sign of forgetting on both Permuted MNIST and Split MNIST.

This work provides many possibilities for future developments, especially about the growth and removal mechanisms.

Regarding the growth mechanism, for example, it would be interesting to explore different neuron diversification method other than dropout. Also defin-

ing an adaptive distance threshold for prototypes insertion that changes based on data distribution would be an intriguing extension.

Instead, the removal policy can be further developed and made more complex, for instance, defining a different importance measure for prototypes other than the number of assigned examples.

Finally, the current growth mechanism can potentially insert an infinite number of prototypes, even applying with the removal mechanism. Thus, limiting the model's memory would be a challenging future development.

Abbreviations

CN-DPM Continual Neural Dirichlet Process Mixture.

CWR Copy Weights with Re-init.

DGN Dendritic Gated Network.

EWC Elastic Weights Consolidation.

ExStream Exemplar Stream.

GLN Gated Linear Network.

GNG Growing Neural Gas.

GR Generative Replay.

GrowPGLN Grow Prototype Gated Linear Network.

HCL Hybrid generative-discriminative approach to Continual Learning.

IGNG Incremental Growing Neural Gas.

MAS Memory Aware Synapses.

NG Neural Gas.

PNN Progressive Neural Network.

ProtoGLN Prototype Gated Linear Network.

SGD Stochastic Gradient Descent.

SI Synaptic Intelligence.

SOM Self-Organizing Map.

SVM Support Vector Machine.

UCB Uncertainty-guided Continual learning with Bayesian Neural Networks.

XdG Context dependent Gating.

Bibliography

[1] R. Aljundi, F. Babiloni, M. Elhoseiny, M. Rohrbach, and T. Tuytelaars. Memory aware synapses: Learning what (not) to forget. *CoRR*, abs/1711.09601, 2017.

[2] R. Aljundi, K. Kelchtermans, and T. Tuytelaars. Task-free continual learning. *CoRR*, abs/1812.03596, 2018.

[3] P. Baldi and P. J. Sadowski. The dropout learning algorithm. *Artif. Intell.*, 210:78–122, 2014.

[4] A. Bifet, B. Hammer, and F.-M. Schleif. Recent trends in streaming data analysis, concept drift and analysis of dynamic data sets. In *ESANN*, 2019.

[5] B. E. Boser, I. Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In D. Haussler, editor, *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992*, pages 144–152. ACM, 1992.

[6] D. Budden, A. H. Marblestone, E. Sezener, T. Lattimore, G. Wayne, and J. Veness. Gaussian gated linear networks. In *NeurIPS*, 2020.

[7] P. Buzzega, M. Boschini, A. Porrello, D. Abati, and S. Calderara. Dark experience for general continual learning: a strong, simple baseline, 2020.

[8] D. G. Clark, L. Abbott, and S. Chung. Credit assignment through broadcasting a global error vector. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[9] G. V. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[10] S. Ebrahimi, M. Elhoseiny, T. Darrell, and M. Rohrbach. Uncertainty-guided continual learning with bayesian neural networks. *CoRR*, abs/1906.02425, 2019.

[11] B. Fritzke. A growing neural gas network learns topologies. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information*

Processing Systems 7, [NIPS Conference, Denver, Colorado, USA, 1994], pages 625–632. MIT Press, 1994.

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

[13] I. J. Goodfellow, M. Mirza, X. Da, A. C. Courville, and Y. Bengio. An empirical investigation of catastrophic forgeting in gradient-based neural networks. In Y. Bengio and Y. LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[14] T. L. Hayes, N. D. Cahill, and C. Kanan. Memory efficient experience replay for streaming learning. *CoRR*, abs/1809.05922, 2018.

[15] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network, 2015.

[16] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[17] D. P. Kingma and M. Welling. Auto-encoding variational bayes. In Y. Bengio and Y. LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[18] P. Kirichenko, M. Farajtabar, D. Rao, B. Lakshminarayanan, N. Levine, A. Li, H. Hu, A. G. Wilson, and R. Pascanu. Task-agnostic continual learning with hybrid probabilistic models. *CoRR*, abs/2106.12772, 2021.

[19] J. Kirkpatrick, R. Pascanu, N. C. 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. *CoRR*, abs/1612.00796, 2016.

[20] T. Kohonen. The self-organizing map. *Neurocomputing*, 21(1-3):1–6, 1998.

[21] B. M. Lake, R. Salakhutdinov, J. Gross, and J. B. Tenenbaum. One shot learning of simple visual concepts. In L. A. Carlson, C. Hölscher, and T. F. Shipley, editors, *Proceedings of the 33th Annual Meeting of the Cognitive Science Society, CogSci 2011, Boston, Massachusetts, USA, July 20-23, 2011*. cognitivesciencesociety.org, 2011.

[22] S. Lee, J. Ha, D. Zhang, and G. Kim. A neural dirichlet process mixture model for task-free continual learning. *CoRR*, abs/2001.00689, 2020.

[23] T. Lesort, V. Lomonaco, A. Stoian, D. Maltoni, D. Filliat, and N. D. Rodríguez. Continual learning for robotics: Definition, framework, learning strategies, opportunities and challenges. *Inf. Fusion*, 58:52–68, 2020.

[24] Z. Li and D. Hoiem. Learning without forgetting. *CoRR*, abs/1606.09282, 2016.

[25] V. Lomonaco. *Continual Learning with Deep Architectures*. PhD thesis, University of Bologna, Italy, 2019.

[26] V. Lomonaco and D. Maltoni. Core50: a new dataset and benchmark for continuous object recognition. *CoRR*, abs/1705.03550, 2017.

[27] D. Lopez-Paz and M. Ranzato. Gradient episodic memory for continuum learning. *CoRR*, abs/1706.08840, 2017.

[28] D. Maltoni and V. Lomonaco. Continuous learning in single-incremental-task scenarios. *CoRR*, abs/1806.08568, 2018.

[29] T. Martinetz, S. G. Berkovich, and K. Schulten. 'neural-gas' network for vector quantization and its application to time-series prediction. *IEEE Trans. Neural Networks*, 4(4):558–569, 1993.

[30] N. Y. Masse, G. D. Grant, and D. J. Freedman. Alleviating catastrophic forgetting using context-dependent gating and synaptic stabilization. *Proc. Natl. Acad. Sci. USA*, 115(44):E10467–E10475, 2018.

[31] C. Mattern. Linear and geometric mixtures - Analysis. *Data Compression Conference Proceedings*, pages 301–310, 2013.

[32] J. L. McClelland, B. L. McNaughton, and R. C. O'Reilly. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review*, 102 3:419–457, 1995.

[33] M. McCloskey and N. J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. volume 24 of *Psychology of Learning and Motivation*, pages 109–165. Academic Press, 1989.

[34] T. M. Mitchell. *Machine learning, International Edition.* McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[35] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71, 5 2019.

[36] G. I. Parisi and V. Lomonaco. Online continual learning on sequences. *CoRR*, abs/2003.09114, 2020.

[37] Y. Prudent and A. Ennaji. An incremental growing neural gas learns topologies. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 1211–1216 vol. 2, 2005.

[38] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, mar 1986.

[39] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[40] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[41] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell. Progressive neural networks, 2016.

[42] E. Sezener, A. Grabska-Barwińska, D. Kostadinov, M. Beau, S. Krishnagopal, D. Budden, M. Hutter, J. Veness, M. Botvinick, C. Clopath, M. Häusser, and P. E. Latham. A rapid and efficient learning rule for biological neural circuits. *bioRxiv*, 2021.

[43] H. Shin, J. K. Lee, J. Kim, and J. Kim. Continual learning with deep generative replay. *CoRR*, abs/1705.08690, 2017.

[44] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[45] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[46] J. Veness, T. Lattimore, A. Bhoopchand, D. Budden, C. Mattern, A. Grabska-Barwinska, P. Toth, S. Schmitt, and M. Hutter. Gated linear networks. *CoRR*, abs/1910.01526, 2019.

[47] J. Veness, T. Lattimore, A. Bhoopchand, A. Grabska-Barwinska, C. Mattern, and P. Toth. Online learning with gated linear networks. *CoRR*, abs/1712.01897, 2017.

[48] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[49] F. Zenke, B. Poole, and S. Ganguli. Continual learning through synaptic intelligence. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 3987–3995. PMLR, 2017.