BACHELOR'S THESIS

Learn, don't forget: constructive methods for effective continual learning

Author:
Aitor GANUZA IZAGIRRE

Supervisor:
Prof. Pascal FROSSARD

Local Tutor:
Prof. Javier RUIZ-HIDALGO

In partial fulfillment of the requirements for the

Bachelor's Degree in Mathematics
Bachelor's Degree in Telecommunication Technologies and Services Engineering

Signal Processing Laboratory (LTS4)
École Polytechnique Fédérale de Lausanne

Lausanne, July 2022

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona
telecos BCN

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de Formació Interdisciplinària Superior

Learn, don't forget: constructive methods for effective continual learning

Aitor GANUZA IZAGIRRE

Abstract

The hallmark of artificial intelligence lies in agents with capabilities to adapt to continuous streams of information and tasks. Continual Learning aims to address this challenge. However, machine learning models accumulate knowledge in a manner different from humans, and learning new tasks leads to degradation in past ones, a phenomenon aptly named *catastrophic forgetting*. Most continual learning methods either penalize the change of parameters deemed important for past tasks (regularization-based methods) or employ a small replay buffer (replay-based methods) that feeds the model examples from past tasks in order to preserve performance. However, the role and nature of the regularization and the other possible factors that make the continual learning process effective are not well understood. The project sheds light on these questions and suggests ways to improve the performance of continual learning in vision tasks such as classification.

Key words: Continual Learning, Lifelong Learning, Online Learning, Artificial Intelligence, Machine Learning

AMS Codes: 68T05 (Learning and adaptive systems in artificial intelligence), 68T07 (Artificial neural networks and deep learning), 68T30 (Knowledge representation).

Aprender sin olvidar: métodos constructivos para un aprendizaje continuo efectivo

Aitor Ganuza Izagirre

Resumen

El objetivo distintivo de la inteligencia artificial reside en conseguir agentes con capacidad para adaptarse a flujos continuos de información. El aprendizaje continuo pretende dar respuesta a este reto. Sin embargo, los modelos de aprendizaje automático acumulan el conocimiento de una manera diferente a la de los humanos, y el aprendizaje de nuevas tareas conduce a la degradación de las pasadas, fenómeno denominado *olvido catastrófico*. La mayoría de los métodos de aprendizaje continuo o bien penalizan el cambio de los parámetros considerados importantes para las tareas pasadas (métodos basados en la regularización) o bien emplean un pequeño búfer de repetición (métodos basados en la repetición) que alimenta el modelo con ejemplos de tareas pasadas para preservar el rendimiento. Sin embargo, el papel exacto que juega la regularización y los demás posibles factores que hacen que el proceso de aprendizaje continuo sea eficaz no se conocen bien. El proyecto arroja luz sobre estas cuestiones y sugiere formas de mejorar el rendimiento del aprendizaje continuo en tareas de visión como la clasificación.

Palabras clave: Aprendizaje Continuo, Aprendizaje Permanente, Aprendizaje en Línea, Inteligencia Artificial, Aprendizaje Automático

Códigos AMS: 68T05, 68T07, 68T30.

Aprendre sense oblidar: mètodes constructius per a un aprenentatge continu efectiu

Aitor GANUZA IZAGIRRE

Resum

L'objectiu distintiu de la intel·ligència artificial és aconseguir agents amb capacitat per adaptar-se a fluxos continus d'informació. L'aprenentatge continu pretén donar resposta a aquest repte. No obstant això, els models d'aprenentatge automàtic acumulen el coneixement d'una manera diferent de la dels humans, i l'aprenentatge de noves tasques condueix a la degradació de les passades, fenomen anomenat *oblit catastròfic*. La majoria dels mètodes d'aprenentatge continu o penalitzen el canvi dels paràmetres considerats importants per a les tasques passades (mètodes basats en la regularització) o bé emprenen una petita memòria intermèdia de repetició (mètodes basats en la repetició) que alimenta el model amb exemples de tasques passades per preservar el rendiment. Tot i això, el paper exacte que juga la regularització i els altres possibles factors que fan que el procés d'aprenentatge continu sigui eficaç no es coneixen bé. El projecte dóna llum sobre aquestes qüestions i suggereix maneres de millorar el rendiment de l'aprenentatge continu en tasques de visió com la classificació.

Paraules clau: Aprenentatge Continu, Aprenentatge Permanent, Aprenentatge en Línia, Intel·ligència Artificial, Aprenentatge Automàtic

Codi AMS: 68T05, 68T07, 68T30.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor Prof. Pascal Frossard for his support throughout my thesis. I also want to thank my advisor Nikolaos Dimitriadis for being the reference person in the laboratory on a daily basis, for his patience in answering any of my questions and for his involvement in the project. Many thanks to all the LTS4 team members for the host.

Of course, I am grateful to CFIS for the financial, personal and professional support, and for giving me this opportunity to finish my degrees in Lausanne.

Lastly, thanks to my family and friends for supporting me in the distance.

Contents

List of Figures

List of Tables

List of Abbreviations

CL	Continual Learning
ML	Machine Learning
DL	Deep Learning
EWC	Elastic Weight Consolidation
SI	Synaptic Intelligence
MAS	Memory Aware Synapses
KFA	Kronecker Factored Laplace Approximation
i.i.d.	Independent and Identically Distributed
RL	Reinforcement Learning
CRL	Continual Reinforcement Learning
MDP	Markov Decision Process
GD	Gradient Descent
SGD	Stochastic Gradient Descent
w.r.t.	with regard to
NN	Neural Network
ANN	Artificial Neural Network
QP	Quadratic Program
GAP	Global Average Pooling

Chapter 1

Introduction

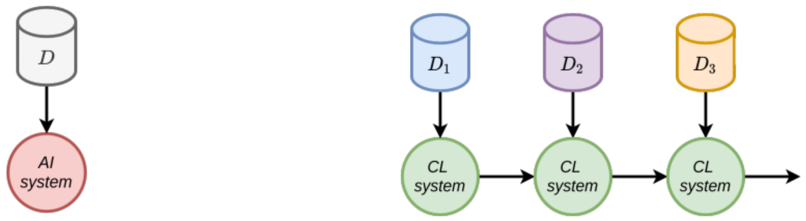
Humans and other animals have the ability to continually acquire, fine-tune, and transfer knowledge without forgetting previously learned information through complex neurocognitive mechanisms.

It is therefore of great interest to emulate on machines this way of learning, with the ultimate goal of having intelligent agents adapted to the constantly changing world. We define Continual Learning (CL) as the ability to learn continually from a stream of data, building on what was learned previously, while being able to reapply, adapt and generalize to new situations. In machine learning terms, this is also known as lifelong learning, sequential learning, or incremental learning. A CL algorithm must be capable of learning from a continuous stream of information, with such information becoming progressively available over time and where the number of tasks to be learned, such as membership classes in a classification task, are not predefined (Parisi et al., 2018).

In *traditional* learning, Figure 1.1 ((Cossu, Ziosi, and Lomonaco, 2021)), whenever a model has finished learning, it remains unchanged when used in practice. It is therefore a static modeling of a continually changing world. The hypothesis of independent and identically distributed distributions (i.i.d.) is necessary for these systems to work properly. See section A.2 for details on this hypothesis. All data samples must come from the same probability distribution, which is invariant over time. Data sets are usually huge and fixed, and the data must be sufficiently shuffled to ensure this condition.

A CL model is required to incrementally build and dynamically update internal representations as the distribution of tasks dynamically changes across its lifetime. Ideally, part of these internal representations will be general and invariant enough to be reusable in similar tasks, while another part should preserve and encode task-specific knowledge (W. ContinualAI, 2021).

The objective of the thesis is to analyze and compare existing CL methods and provide information on how to learn effectively without forgetting. This thesis is structured as follows.



(A) Static system (B) Continual learning system

FIGURE 1.1: (Cossu, Ziosi, and Lomonaco, 2021). While offline learning learns in a single phase from a static set of data, Continual Learning systems learn from a stream of non-stationary data.

- **Chapter 2** introduces and details all the needed terms specific to the CL field. It describes the main objectives, constraints, limitations, and tools of Continual Learning. This chapter aims to make the reader understand the context of Continual Learning, and its situation w.r.t. the AI community.
- **Chapter 3** explains some of the actual methods on Continual Learning. They are divided into three main categories: regularization based methods, memory-replay based methods, and parameter isolation based methods. The advantages / disadvantages, intuitive justifications, and mathematical derivations are provided. Some of these methods have been implemented as part of the thesis, and they are used in the experiments.
- **Chapter 4** describes the framework developed on Python to act as a Continual Learning solver. It implements multiple methods and allows the user to modify all settings and hyperparameters, with the objective of being as flexible as possible for experiments.
- **Chapter 5** describes the experiments made on Continual Learning. They have been developed using the chapter 4 solver, modifying the algorithms, and observing possible improvements.
- **Chapter 6** briefly summarizes and concludes the thesis, suggesting possible future directions.

Chapter 2

Background and general introduction to Continual Learning

2.1 Desiderata

A possible solution to Continual Learning might be saving all data, shuffling them to ensure i.i.d hypothesis (section A.2), and training it with a traditional learning setting. However, this is neither always possible nor optimal. Typical constraints include limited memory that cannot store the data, privacy issues related to the storage of the samples, or time considerations such as having to use the model while still learning. In the following (nonexhaustive) settings, a Continual Learning approach is necessary:

- Having a pre-trained model, you want to update it with new different data, but you do not have access to the original training data any longer. If the objective was to simply learn the new incoming data without regard for preserving prior knowledge, it would be a transfer learning problem (Zhuang et al., 2019). However, if we aim to perform well both on previous and current data distributions, it is a Continual Learning problem.
- You want an agent to learn a different policy or task, but there is no information on when or how the learning objective changes.
- You want to learn from a continuous stream of data, and you want to have your model updated at any given time (so it is not possible to simply store all the data and train it later), and the data might change over time.

To handle such settings, Continual learning is characterized in practice by a series of **desiderata** (Cossu, Ziosi, and Lomonaco, 2021):

- No access to previously encountered data. However, this assumption is relaxed in memory-replay based methods (section 3.3), where re-visiting old data is allowed using a small fixed memory.

- **Efficiency**. Constant computational and memory resources, even if the stream of data and the number of different tasks increase over time. This forces the system to use its resources intelligently.
- **Scalability**. Incremental development of ever more complex knowledge and skills.
- The learning agent should be able to transfer and adapt what it learned from previous experience, data, or tasks to new situations, and make use of more recent experience to improve performance on the capabilities learned earlier.

2.2 Sustainability of AI

Artificial Intelligence is becoming more and more present and important in our society, and our awareness of the ethical issues related to the design and development of AI systems raises at the same time (Cossu, Ziosi, and Lomonaco, 2021). The term AI ethics refers to the study of ethical and societal issues facing developers, producers, consumers, citizens, policymakers, and civil society organizations (Wynsberghe, 2021). Most of the public debate on the ethics of AI is about concerns about the application of AI techniques. They are expressed by values such as fairness, privacy, accountability, transparency, etc. However, the focus on the importance of sustainability is something that has not been addressed as much within the framework of AI ethics. Approaching artificial intelligence through the Continual Learning perspective is definitely a way of both addressing these values and promoting sustainability in the field. CL can provide environmentally and financially sustainable solutions.

The advancements in both hardware (more powerful GPUs) and software (deep learning models, open-source frameworks and supporting libraries) have significantly improved the accuracy and training time of Artificial Neural Networks. However, the high speed and accuracy are at the cost of energy consumption. The more the size of datasets grows, the more the energy demand for training such datasets increases. It is highly desirable to design deep learning frameworks and algorithms that are both accurate and energy efficient (D. Li et al., 2016).

There has been more effort to analyze the principles, 'what' of AI ethics, rather than on practices, the 'how' (Morley et al., 2020). Continual Learning can be a suitable candidate to convert current AI ethics principles to practices and to be more socially, financially and environmentally sustainable.

Learning continually means not only being able to adapt to unpredictable circumstances not foreseen during the design of our AI systems, but also to build systems that are efficient, scalable and easily amendable, or in other words, sustainable.

The advantage in efficiency and scalability for Continual Learning with respect to offline AI models increases with the non-stationary nature of the environment: in *traditional* learning, each new piece of information may require to retrain a model on both new and previous data. However, most of the knowledge needed is already inside the model. Continual learning, on the contrary, only requires training a model on the new incoming samples, possibly exploiting temporal correlations between successive patterns to improve performance.

2.3 Catastrophic forgetting

All natural cognitive systems gradually forget previously learned information. Plausible models of human cognition should therefore exhibit similar patterns of gradual forgetting of old information as new information is acquired. Natural cognitive systems do not, in general, forget *catastrophically* (all of a sudden). Unfortunately, this does not happen in artificial neural networks (French, 1999). Due to the sequential nature of the continual learning paradigm, catastrophic forgetting is the main issue to be addressed.

Catastrophic forgetting, also known as catastrophic interference, is the tendency of an artificial neural network to completely and abruptly forget previously learned knowledge when learning new information (McCloskey and Cohen, 1989).

This is a clear example of the *stability-plasticity* dilemma, also known as the *sensitivity-stability* dilemma. Learning requires plasticity for the integration of new knowledge, but also stability to prevent forgetting of previous information. Too much plasticity leads to constantly forget previously encoded data, i.e., catastrophic forgetting. However, too much stability impedes the efficient coding of new data.

When working with labeled tasks, this phenomenon becomes evident, as in Figure 2.1 (Hong, Y. Li, and Shin, 2019), where an agent tries to sequentially learn five different tasks and evaluates the performance on all the tasks learned so far. When learning a new task starts, performance on previously learned tasks *catastrophically* drops.

In the concrete case of an artificial neural network trying to learn sequentially two different tasks with different distributions, a standard optimization algorithm such as Stochastic Gradient Descent will first converge to a solution where the loss function is low for the first task. Then, if we train this network with the second task, the weights will change in order to minimize the loss in the second task. This could lead to leaving the region where the loss is low for the first task, as shown in Figure 2.2 by Kirkpatrick et al., 2016 in their Elastic Weight Consolidation (EWC) method, addressed in subsection 3.2.1.

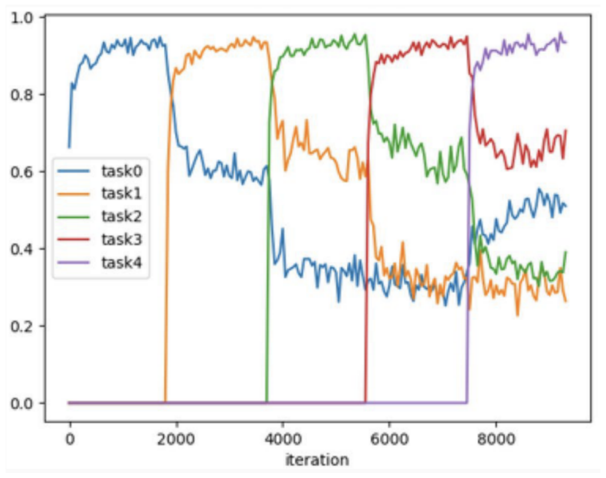


FIGURE 2.1: (Hong, Y. Li, and Shin, 2019). Catastrophic forgetting. The accuracy on five different tasks (each with a different color) is plotted. At the beginning (first iterations), task0 is learned. When the learning for this task is complete, it starts for task1, causing the blue line to drop suddenly and significantly. This happens whenever we start to train on a new task.

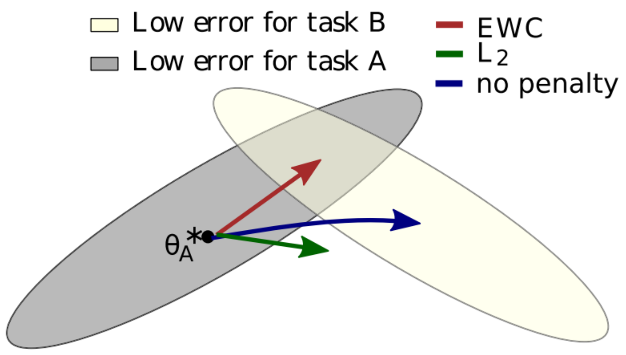


FIGURE 2.2: (Kirkpatrick et al., 2016). EWC method ensures that task A is remembered while training on task B. Training trajectories are illustrated in a schematic parameter space, with parameter regions leading to good performance on task A (gray) and on task B (cream color). After learning the first task, the parameters are at a local optimum for task A, θ_A^* . If we take gradient steps according to task B alone (blue arrow), we will minimize the loss of task B but destroy what we have learned for task A (too much *plasticity*). On the other hand, if we constrain each weight with the same coefficient (green arrow), the restriction imposed is too severe and we can remember task A only at the expense of not learning task B (too much *stability*). The objective would be to find a low-loss area for task B without incurring a significant loss on task A (red arrow)

2.4 Implications on main machine learning paradigms

In this section, the relationship of continual learning with the main machine learning paradigms is discussed. Supervised learning, unsupervised learning, and reinforcement learning are linked to continual learning in different ways.

2.4.1 Supervised Continual Learning

Supervised learning is the machine learning task of learning a function that maps an input to an output based on examples of input-output pairs (Russell, 2010a). It infers a function from labeled training data consisting of a set of training examples. In the CL paradigm, each data sample from the streaming is a tuple $\langle x_k, y_k \rangle$ of input and target. If tasks are labeled, the data sample is $\langle x_k, y_k, t_k \rangle$, where t_k is the task identifier.

Supervised continual learning is the easiest setup to obtain good performance in a problem. Here, relatively simple and easy to implement continual learning methods can be applied and still obtain very revealing results. Most efforts in supervised continual learning are not in the direction of solving state-of-the-art machine learning problems, but rather to gain a better understanding of how continual learning algorithms work, and to delve into the whys and wherefores of catastrophic forgetting and how the encoding of old and new data happens. In fact, this is the reason why this thesis is mainly focused on supervised continual learning.

Classification problems are the most popular scenarios for delving into these phenomena. In fact, image-classification benchmarks, discussed in section 2.9 are the main baselines for this purpose.

2.4.2 Unsupervised Continual Learning

Unsupervised Learning is a paradigm that analyzes and clusters unlabeled datasets. These algorithms are used to discover hidden patterns without the need for human intervention (IBM, 2020), and without any labels or rewards to learn from. While learning, an unsupervised network tries to mimic or reproduce the given data and uses the error in this generated output to correct its weights and biases.

In Unsupervised Continual Learning, the focus is on learning representations without any knowledge about task identity, and exploring scenarios with both abrupt changes between tasks and smooth transitions from one distribution to another (Rao et al., 2019). That is, a very general task-agnostic approach to continual learning. It can be applied to multiple fields of knowledge. For example, Hemati, Schreyer, and Borth, 2021 use Continual Learning techniques for unsupervised anomaly detection in continuous auditing of financial accounting data. Allred and Roy, 2020 is a bioinspired continual learning approach where they use CL techniques on Spike Neural Networks (SNN)

(Maass, 1997) to adapt to novel information while protecting essential knowledge from catastrophic forgetting.

This is a far more complex setting than supervised continual learning, and there is an increasing interest in adapting and applying all the knowledge, methods, and algorithms from supervised learning to this area. In fact, Artificial General Intelligence (Russell, 2010b) is a very general case of unsupervised continual learning.

2.4.3 Continual Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm in which the goal is to train an agent to perform actions in a particular environment in order to maximize the expected cumulative reward.

It is formalized as a Markov Decision Process (MDP) defined by a tuple $\langle \mathcal{S}, \mathcal{A}, p_s, r \rangle$, where at time step t, the RL agent observes the state $s_t \in \mathcal{S}$, takes an action $a_t \in \mathcal{A}$, which results in a reward $r(s_t, a_t)$ and transition to next state s_{t+1} with probability $p_s(s_{t+1}|s_t, a_t)$. The objective of the agent is to find a policy defined by a probability distribution over actions given the state $\pi(a_t|s_t)$, that maximizes its expected sum of future rewards:

$$\pi^* = \text{argmax}_\pi \sum_t \mathbb{E}_\pi[r(s_t, a_t)]$$

where $\mathbb{E}_\pi$ is the expectation under the reward distribution defined by policy π.

In traditional Reinforcement Learning, the objective is to learn an unknown stationary MDP, that is, fixed states and rewards. However, as real-world problems are non-stationary, complex problems in Reinforcement Learning could be approached in a continual learning way. In fact, nowadays Reinforcement Learning and Continual Reinforcement Learning are often mixed. This is because in RL data are typically accumulated online as the agent interacts with the environment, the distribution of experiences is often non-stationary over the training of a single task, as well as across tasks, since (*i*) experiences are correlated in time and (*ii*) the agent's policy changes as it learns. In Kaplanis, Shanahan, and Clopath, 2018, they propose a way to mitigate catastrophic forgetting in the RL environment without the need of a replay database, and to be a bit closer to how the brain achieves continual learning.

Even if most of the bibliography tests their Continual Learning algorithms in a supervised manner, there is also interest in trying reinforcement learning benchmarks, such as the Atari games (Mnih et al., 2013), which for example were used by Kirkpatrick et al., 2016.

2.5 Definition and framework

Continual Learning is a very broad field, where the problem and the way to tackle it completely change depending on the constraints of each case. Despite the growing interest in CL in recent years (Parisi et al., 2018), very little effort has been devoted to a common formalization of algorithms that learn continually in dynamic environments. Therefore, every problem should be analyzed individually. One must clearly describe (Lesort et al., 2019):

- **Data availability**. Are data distributions assumed i.i.d. at any point (clearly distinguished tasks)? Is each task encountered only once?
- Assumed **prior knowledge**. Is the algorithm agnostic with respect to the structure (number of classes in a classification problem, number of tasks, etc.) of the data stream? Is it necessary to start from a pre-trained model? If so, which is the already learned information, and which is the expected information to continually learn?
- **Memory and computation** requirements. How much available memory is required while learning? Is there any computational overhead added over time? Is the algorithm able to handle real-time situations where there is not enough time to learn? In some cases, the agent is required to perform in a very short time since the availability of the data.
- **Type of supervision**. In the case of multiple tasks, is the task label available to the agent during training / evaluation?

A specific CL framework could vary from a general unsupervised learning problem without the notion of tasks to a supervised classification problem where each data point has a task-id. In order to help answer the proposed questions and formalize the framework, we can formulate some definitions.

In the list below (Hsu, Y. Liu, and Kira, 2018), a non-exhaustive list of CL scenarios (possible combinations of answers to some of these questions) is provided. For notation, $\mathcal{X}_1$ is the input set of the old task, where $x_k^1 \in \mathcal{X}_1$ are the input data samples, and $\mathcal{X}_2$ is the input set of the new task. Similarly, $\mathcal{Y}_1$ and $\mathcal{Y}_2$ correspond to the target sets of the old and new tasks, respectively.

- **Domain Incremental Learning**: The input space (domain) is incremental, which means that $P(\mathcal{X}_1) \neq P(\mathcal{X}_2)$. However, the output space remains the same, $\mathcal{Y}_1 = \mathcal{Y}_2$. Therefore, the task identifier is not needed. The model has to learn incrementally the distributions of each task. As an example, we could have $\mathcal{X}_1 =$ images of cats/dogs in the garden, $\mathcal{X}_2$ = images of cats/dogs indoors, $\mathcal{Y}_1 = \mathcal{Y}_2 = \{cat, dog\}$.

The objective remains *distinguishing cats and dogs*, but the distribution of the domain changes at the new task. However, the agent does not receive a task identifier.

- **Class Incremental Learning**. This scenario relates to multiclass classification. Here, $P(\mathcal{X}_1) \neq P(\mathcal{X}_2)$, but $\mathcal{Y}_1 \cap \mathcal{Y}_2 = \varnothing$. This means that each task in the sequence contains an exclusive subset of classes in a dataset. As task identifiers are not provided, the agent must be able to classify between all classes learned so far. Continuing with the example of the previous scenario, here in the first task we would have images of cats and dogs, whereas in the second task we have images of cows and horses, and $\mathcal{Y}_1 = \{cat, dog\}$, and $\mathcal{Y}_2 = \{cow, horse\}$.
- **Task Incremental Learning** Here the setting is very similar to Class Incremental Learning, but a task identifier t is provided. In this scenario, a multi-head architecture with one head for each task can be used to exploit the information about task identifier. In the forward pass, only the head matching t is activated to make predictions. For further details on multi-head classifiers, view section 2.10.

2.6 Formalization of a classical CL problem

In this section, a *classic* continual learning scenario is described. This thesis was developed under these assumptions. In a classification problem, which is a type of Supervised Continual Learning problem (subsection 2.4.1), a finite number of tasks is learned from a sequence of experiences $S = (e_1, e_2, \ldots, e_n)$. Each experience is assumed to have its data independent and identically distributed, and it is learned as an atomic unit. Within the learning of an experience, it is usually allowed to revisit samples from the experience and learn in multiple epochs. The setting where each experience corresponds to a unique labelled task is called **task-incremental learning**. In this case, the number of tasks $T = n$. Data from each experience $\mathcal{D}^i$ is split in the following way: $\mathcal{D}^i = \mathcal{D}^i_{train} \sqcup \mathcal{D}^i_{val}$, usually around 80% for training and 20% for validation. The datasets used in the experiments (chapter 5) come from predefined splits.

Each sample from $\mathcal{D}^i$ is a tuple $\langle x^i_k, y^i_k, t^i_k \rangle$. In our setting, the agent knows the task identifier $t^i_k \in \mathcal{T}$ (where $\mathcal{T} = 1, \ldots, T$) only in some experiments. In others, what the agent "sees" is only $\langle x^i_k, y^i_k \rangle$. This distinction is crucial in terms of performance, as having the task-id information allows the architecture to be multi-head, that is, to activate a different decoder depending on the task.

The input space is $\mathcal{X} = \mathcal{X}_1 \sqcup \mathcal{X}_2, \sqcup \ldots, \sqcup \mathcal{X}_T$, and each $x^i_k \in \mathcal{X}_{t^i_k}$. Note that, in general, $t^i_k = i$ in task-incremental learning (this will always be the case in our setting).

The output space is $\mathcal{Y}$. In **class-incremental learning**, which is an orthogonal term to task-incremental learning, $\mathcal{Y} = \mathcal{Y}_1 \sqcup \mathcal{Y}_2 \sqcup \ldots \sqcup \mathcal{Y}_T$, which means that the output is task-specific. If the agent knows the task identifier t beforehand, it will only have to classify among the classes in $\mathcal{Y}_t$ instead of $\mathcal{Y}$. If the learning is not class-incremental, the output can be of any class despite the task-id.

A Continual Learning algorithm $\mathcal{A}^{CL}$ is a function with the following input and output spaces (Carta et al., 2021):

$$\left(f_{i-1}^{CL}, \mathcal{D}_{train}^{i}, \mathcal{M}_{i-1}\right) \longrightarrow \left(f_i^{CL}, \mathcal{M}_i\right)$$

where f_i^{CL} represents the model after having learned from experience i, and $\mathcal{M}_i$ is a memory storing knowledge from previous experiences. This memory could be either samples seen previously or other kind of processed information. Methods categorized at section 3.3 use these memories in different ways, but the other methods have $\mathcal{M}_i = \varnothing \; \forall i$.

The objective of $\mathcal{A}^{CL}$ is to minimize the total loss of the stream $\mathcal{L}_S$ on the entire data set $\sqcup_i \mathcal{D}_{train}^i$. In section 2.7 some useful metrics are described.

$$\mathcal{L}_S(f_i^{CL}, n) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_{exp}(f_i^{CL}, \mathcal{D}_{val}^i),$$

where

$$\mathcal{L}_{exp}(f_i^{CL}, \mathcal{D}_{val}^i) = \frac{1}{|\mathcal{D}_{val}^i|}\sum_{j \in \mathcal{D}_{val}^i} \mathcal{L}(f_i^{CL}(x_j^i), y_j^i)$$

Note that depending on the experiment, we might have a different objective. For example, $\mathcal{L}_{exp}$ could also be task-specific: $\mathcal{L}_{exp}^i$, but in this case we are considering classification for all tasks, $\mathcal{L}_{exp}^i = \mathcal{L}_{exp} \forall i \in \{1, \ldots, n\}$.

The loss $\mathcal{L}(f_i^{CL}(x_j^i), y_j^i)$ is computed on a single sample, usually using cross-entropy loss for classification tasks. For a detailed explanation of the cross-entropy loss, see section A.1.

2.7 Metrics

In order to compare different algorithms, it is essential to have the correct metrics. Measuring the performance and accuracy of algorithms with intelligent metrics can be a determining factor when having to decide which approach is *better* among many. Different ways of measuring can tell completely different things about an experiment, so it is crucial to understand which metrics are more informative at each case.

As the nature of a Continual Learning setting is different from the classical machine learning problems, it is particularly interesting to define specific metrics for measuring the performance of a CL algorithm. CL is a more *human-like* approach to artificial intelligence, so it is useful to have defined ways to measure concepts like forgetting or knowledge transfer.

Lopez-Paz and Ranzato, 2017 were the first to take this into account and propose some metrics related to this knowledge transfer among tasks.

For a principled evaluation, let's consider the set of tasks $\mathcal{T} = \{1, \ldots, T\}$. After the agent finishes learning task t, we evaluate its *validation* performance on all T tasks. In this way, we construct a matrix $R \in \mathbb{R}^{T \times T}$, where $R_{i,j}$ is the classification accuracy of the model after observing the last sample from task i and evaluating it on task j. Note that a high R_{ii} means good learning ability or *plasticity*, and a high R_{ij} of $i > j$ means good retention ability or *stability*. For $i < j$, the evaluation is made in a task that has not been learned yet. In general it is not very interesting, except for some specific metrics such as Forward Transfer, explained below.

For normalization purposes, we also consider b_i the accuracy for task i at random initialization of the model. We define the following metrics:

$$\textbf{Average Accuracy: } \mathrm{ACC} = \frac{1}{T} \sum_{i=1}^{T} R_{T,i}$$

$$\textbf{Average Backward Transfer: } \mathrm{BWT} = \frac{1}{T-1} \sum_{i=1}^{T-1} R_{T,i} - R_{i,i}$$

$$\textbf{Average Forward Transfer: } \mathrm{FWT} = \frac{1}{T-1} \sum_{i=1}^{T} R_{i,i+1} - b_{i+1}$$

We can also refer to the Backward Transfer between tasks $i < j$ as $R_{j,i} - R_{i,i}$, and Forward Transfer of task $i \in \{1, \ldots, T-1\}$ as $R_{i,i+1} - b_{i+1}$.

The larger these metrics, the better the model. Depending on the specific purpose, we might want to focus more on one metric or another.

Average Accuracy is the most basic metric of a CL algorithm. It shows the mean performance evaluating on all tasks after having finished to learn all experiences. In general, it is the most informative metric.

Average Backward Transfer or forgetting. It measures the performance of the fully trained model in comparison to how accurate the model was on each task just after having learned it. If this metric is close to zero, it means that the algorithm *does not forget* how to solve past tasks. If it is negative, it means that forgetting is happening. This metric can also be positive. In this ideal case, the algorithm at the end performs even better on task i than when having trained that task. This could happen when different tasks are related.

Average Forward Transfer measures if the model has any kind of skill to solve next task before starting to train on it. This is why it compares the accuracy on a future task with accuracy having a random initialization. The interest of this metric is to see if previous experiences are beneficial in solving future tasks. Note that having Forward Transfer $\longrightarrow$ new task is related to previous tasks, but the opposite implication is not true. It can happen, for instance, in Domain Incremental settings, where even if $P(\mathcal{X}_1) \neq P(\mathcal{X}_2)$, knowing the $P(\mathcal{X}_1)$ distribution helps to solve the task 2. However, in class-incremental scenarios it is unlikely to have this metric and thus it might be not of great interest.

Serrà et al., 2018 propose a different measure called **forgetting ratio** to compare between different task accuracies. This also takes into account a comparison of the CL accuracy with the accuracy that we would get when jointly training in a multitask way (explained in section 2.8).

$$\rho^{j\leq i} = \frac{R_{i,j} - A_i}{J_{i,j} - A_i} - 1,$$

where $J_{i,j}$ is the accuracy measured on task i after jointly learning j tasks in a multitask fashion. A_i is the accuracy of a random stratified classifier using the class information of task i. A stratified classifier generates random predictions by respecting the class distribution of the training data. Note that $\rho \approx -1$ and $\rho \approx 0$ correspond to performances close to those of the random and multitask classifiers, respectively. For having a single metric after learning i tasks, we take the average $\rho^{\leq i} = \frac{1}{i}\sum_{j=1}^{i} \rho^{j\leq i}$.

Hayes et al., 2018 they describe a metric used to evaluate *plasticity* of the CL algorithm in comparison to multi-task learning:

$$\Omega = \frac{1}{T}\sum_{i=1}^{T}\frac{R_{i,i}}{J_{i,i}}$$

Note that in multi-task learning, the effort made to minimize the loss of all tasks is the same. This means that if the difficulty of the tasks is similar, $J_{i,i}$ and $J_{j,i}$ should be similar for all $j < i$. This is not true in the case of R, where if *plasticity* is high enough, a major part of the network's capacity could be used for specific knowledge about task i. Thus, if the network capacity is not high enough to have a good joint average accuracy, it could happen that $\Omega > 1$. This may suggest that a larger architecture should be used.

2.8 Baselines

In order to understand metric reports (explained in the previous section) it is convenient to have some baseline experiments to compare with. These baselines can be used for having a notion of lower and upper bounds. In fact, in the previous section, some of the metrics used the results of these baselines to be more understandable without the context of the experiment.

- **Naive learning** or **fine-tuning**. It is simply doing backpropagation, usually by Stochastic Gradient Descent. The only goal when having a new task is to converge to its minimum loss without paying attention to what happens with previous tasks and with no countermeasures to preserve past performance. It is interesting to have this as a baseline to:
  - See when and how catastrophic forgetting occurs. Depending on choices such as the architecture, the way the network forgets with fine-tuning can vary a lot, as explained in section 2.10. Fine-tuning could be a lower bound for the accuracy on previous tasks.
  - Evaluate the maximum *plasticity* of the network. Note that this is not an upper bound for accuracy in a new task. A way to illustrate this could be the following setting: a network that is already able to solve 2 tasks might have some advantage when finding a better minimum on the loss of a third task (if the third task is very similar to the first but different to the second) than a fine-tuned network which is only able to solve the second task.
- **Cumulative** or **joint training**. For every experience, all data is accumulated, shuffled and re-trained from scratch until convergence. In this way, all the inconveniences that Continual Learning algorithms face disappear. Cumulative training could be colloquially described as *the best you can do with all the data until experience i starting from scratch*. Therefore, it is an upper bound on the CL algorithms. Joint training on the last experience, that is, training with all data at the same time, is also known as **multitask learning**.

2.9 Common Datasets and Benchmarks

When conducting Continual Learning experiments, it is crucial to design an appropriate scenario to evaluate the techniques. Also, agreeing in the community to standardize some benchmarks is very useful to easily compare models. Each of these *canonical* CL benchmarks has its own properties, hence understanding how and when to use them is interesting to extract as much information as possible about the behavior of the algorithms.

In contrast to *traditional ML* benchmarks, in the Continual Learning paradigm one must not confuse dataset and benchmark.

- A Continual Learning **dataset** is the same as a *traditional ML* dataset. It is a collection of data that can be treated as a single unit for analytical purposes.
- A Continual Learning **benchmark** consists of a dataset plus the definition of how to learn from a dataset. This includes how to split the data in different learning experiences, what transformations to apply to the data at each of the experiences,

	Domain Incremental	Class Incremental	Task Incremental
Split MNIST		with or without task-ids	
Permuted MNIST	without task-ids		with task-ids
Rotated MNIST	without task-ids		with task-ids
CIFAR		iCIFAR with or without task-ids	
ImageNet	Non-stationary MiniImagenet		
CORe50	Was designed for this scenario		

TABLE 2.1: Summary of the typical use cases of some continual learning benchmarks.

the number of learning tasks, etc. However, in the bibliography multiple different experiments are commonly called by the same name, even if the scenario defined in section 2.5 is not exactly the same. Therefore, it is crucial to read all the specifications of the definition of the benchmark at each paper, as the nature of the experiment might change a lot.

Some of the most used and most interesting benchmarks for image classification are detailed below. Table 2.1 summarizes the information on the scenarios in which these benchmarks can be applied.

2.9.1 Variants of MNIST dataset

The MNIST database (Lecun, 1998) of handwritten digits has a training set of 60,000 examples and a test set of 10,000 examples of size 28x28 grayscale pixels. The ten digits (0 to 9) have been size-normalized and centered in a fixed-size image. This database is often chosen to develop learning techniques and pattern recognition methods on real-world data while putting minimal effort into preprocessing and formatting.

Split MNIST

This benchmark was initially introduced in a multiheaded form, where the ten digits are split into five two-class classification tasks (Hsu, Y. Liu, and Kira, 2018): $(\mathcal{D}_1, \ldots, \mathcal{D}_5) = (\{\textit{Zeros and ones}\}, \ldots, \{\textit{eights and nines}\})$. Here, the model has five output heads, one for each task, and the task-id is usually known by the model. This constitutes a task-incremental learning (section 2.5) scenario. This dataset was explored in the Synaptic Intelligence paper by Zenke, Poole, and Ganguli, 2017, explained in subsection 3.2.2. Note that the variant of not having task-ids, which is not addressed in our experiments, makes the problem class-incremental.

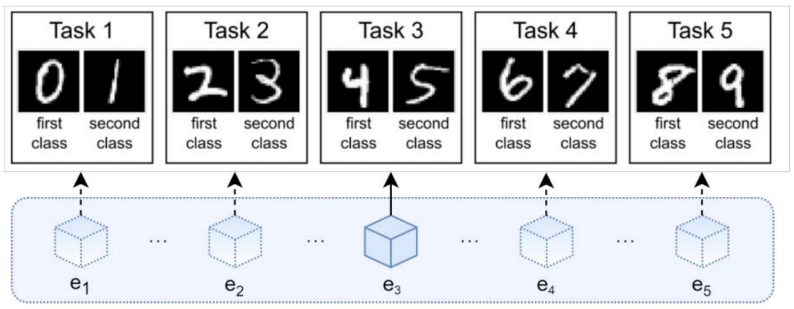


FIGURE 2.3: Split MNIST (C. ContinualAI, 2021)

This scenario is considerably easy to solve, since the selection of the output head is given by the task-id. Another possible scenario of the split MNIST is the single-headed version, proposed by Farquhar and Gal, 2018. In this case, the model is not provided with task-id information (class-incremental learning), therefore it is more challenging.

Permuted MNIST

This benchmark was introduced by Goodfellow et al., 2015. First, a model is trained on MNIST as $\mathcal{D}_1$. The next $\mathcal{D}_i$ for $i > 1$ are constructed from the MNIST images, with fixed random permutations on the pixels of the images and applying the same permutation to images of the same class (Figure 2.4). After training on a new task (new permutation), the evaluation is performed on all tasks. Following the definitions on different scenarios, this would be domain-incremental learning.

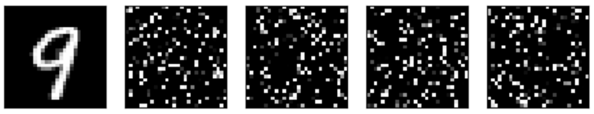


FIGURE 2.4: 5 different permutations of the same image. These permutations are used to create 5 different tasks.

Although it has become a mainstay for continual learning evaluation, this benchmark has been criticized for being too unrealistic. An image with randomly permuted pixels is unrecognizable for humans, as the actual world is not structured like this. In the permuted setting, no example from $\mathcal{D}_t$ looks remotely like an example from $\mathcal{D}_{t-1}$. Hence, there are no correlations between tasks, and concepts like Forward Transfer cannot be studied.

Rotated MNIST

This experiment is a simpler and more realistic version of domain-incremental learning with the MNIST handwritten digits dataset. Each new task consists of a fixed angle rotation of the image. Thus, forward transfer is possible, as tasks are not that different one from another. This could ideally enable an angle-invariant internal representation of the classes.

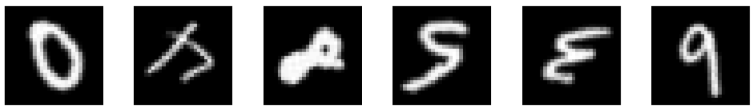


FIGURE 2.5: Images from MNIST dataset rotated at different angles. Each rotation angle corresponds to a different task.

2.9.2 Incremental or Split CIFAR

This is a more complex version of the experiment described in Split MNIST. In the bibliography, they also refer to this benchmark as iCIFAR (Maltoni and Lomonaco, 2019). It can also be used in a task-incremental or class-incremental scenario. CIFAR-10 dataset consists of 32x32 color images in 10 classes. CIFAR-100 dataset is similar to CIFAR-10, except it has 100 classes (Krizhevsky, 2009). These 100 classes are divided into 20 superclasses of 5 classes each. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs).

With these datasets, the following set-ups are the most common: **iCIFAR-10** divides the 10 classes into 5 experiences of 2 classes each. **iCIFAR-100** usually divides the CIFAR-100 dataset into 20 tasks of 5 classes each, where the classes of each task are randomly chosen without taking into account the superclasses. For example, this is used at Lopez-Paz and Ranzato, 2017.

Another possible benchmark is iCIFAR-110, which consists of using CIFAR-10 images as first task, and splitting CIFAR-100 by classes in 10 datasets, having images from 10 classes each. This way, $\mathcal{D}_1$ has CIFAR-10 images and $\mathcal{D}_2, \dots \mathcal{D}_{11}$ is conformed by CIFAR-100 images. CIFAR-10 dataset has more images per class than CIFAR-100, so one has to bear in mind that the number of samples of every task should be the same. This benchmark is used at Zenke, Poole, and Ganguli, 2017.

2.9.3 ImageNet

Split MiniImageNet splits MiniImageNet dataset, which is a subset of ImageNet (Deng et al., 2009) with 100 classes split into 20 disjoint tasks. Each class has 500 colored 84x84 images for training and 100 images for testing. Due to the size of the images, it is an even more complex scenario than split CIFAR, but with very similar characteristics.

Tiny ImageNet is another similar subset of the dataset in the famous ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The dataset contains 100,000 images of 200 classes (500 for each class) downsized to 64×64 colored images. The split TinyImageNet benchmark used in chapter 5 contains 10 classes per task and 20 tasks by default.

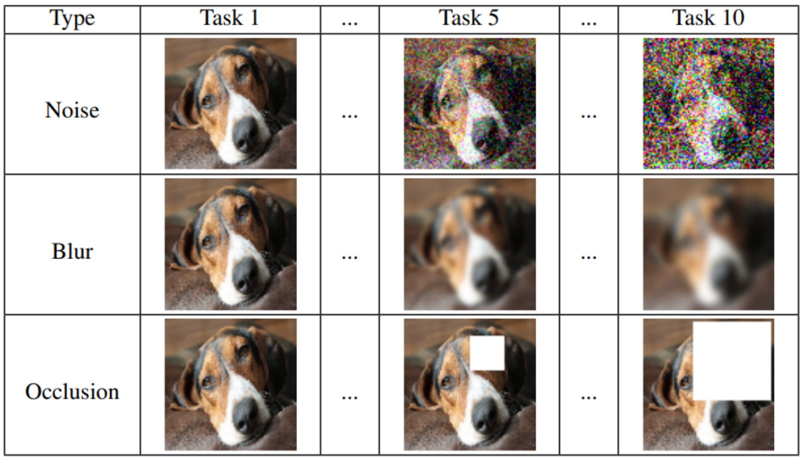


FIGURE 2.6: (Mai et al., 2021). NonStationary-MiniImagenet

NonStationary-MiniImagenet

This benchmark, proposed by Mai et al., 2021, aims to evaluate the domain-incremental setting in a more practical scenario. The reason is that most popular domain-incremental benchmarks are still based on the MNIST dataset, which might be too simple in some cases.

Three non-stationary transformations are applied to the images: noise, blur and occlusion. The *strength* or difficulty added increases over time, as shown in Figure 2.6 (Mai et al., 2021). This way, the agent learns more complex and difficult versions of the *same* skill, and this is one of the objectives of Continual Learning.

2.9.4 CORe50

CORe50 (Lomonaco and Maltoni, 2017), specifically designed for (C)ontinuous (O)bject (Re)cognition, is a collection of 50 domestic objects belonging to 10 categories: plug adapters, mobile phones, scissors, light bulbs, cans, glasses, balls, markers, cups, and remote controls. Objects are hand held in the images, and object occlusions are often produced by the hand itself. Every object is recorded in both outdoors and indoor environments, thus creating a domain-incremental scenario. The images are colored and sized 128x128, so this dataset is the most complex and close to real-world environment described so far.

2.10 Architectures

Most of the research in Continual Learning and the catastrophic forgetting phenomenon have addressed the problem from an algorithmic perspective, with almost no attention dedicated to the architecture used in the experiments. These algorithms are designed to be as robust as possible to the typical distribution shifts in continual learning. The results of the experiments are usually shown using a single architecture, and the reason for choosing that architecture is rarely explained. Architectural decisions such as number of neurons at each layer is usually justified as *the one having better results without compromising too much the efficiency*, or *the same architecture as used by X other method in order to make fair comparisons*. It is not usual to provide further description about the chosen architecture in CL papers, not even in the ones having dynamic architectures for avoiding catastrophic inference.

However, Seyed-Iman Mirzadeh, Chaudhry, et al., 2022 show that the performance of such algorithms is architecture-dependent. A simple example that they provide is Figure 2.7 (Seyed-Iman Mirzadeh, Chaudhry, et al., 2022), where they show that with a fixed architecture, Continual Learning algorithms improve over naive fine-tuning, but with a simple modification in the architecture, such as removing global average pooling, naive fine-tuning turns out to be better. This shows that architectural choices play a key role in Continual Learning.

A relevant finding of S. I. Mirzadeh et al., 2021 is that for Continual Learning, wide networks work far better than narrow networks. Overly deep networks are more prone to catastrophic forgetting than networks overparameterized with more neurons at each layer (and fewer layers). They provide an intuition for this, based on kernel methods: For overparameterized networks, the ℓ_2 distance that the model parameters move during SGD training can be small. In papers connecting learning in NNs to kernel methods, they show that kernels behave like layers of infinite width (Jacot, Gabriel, and Hongler, 2018), and at this limit the kernel does not change throughout the training. Furthermore, Chizat, Oyallon, and Bach, 2018 show that these networks operate in a lazy regime where the norm of the difference between the optimal weights after training at task i and task $i-1$, $||w_i^* - w_{i-1}^*||_2$, is very small. Since wider networks are closer to the infinite width limit, the moving distance may be smaller for wider networks, and thus forgetting is less severe.

The current state of the art on machine learning is held by very deep models, as they are capable of finding very complex patterns due to multiple layers. Therefore, it is a challenge for the future of continual learning to be able to construct deeper models without forgetting. An experiment made on this idea is explained at section 5.2

A **multilayer perceptron** (MLP-N) represents a fully-connected class of feedforward neural network with layers of width N. Being fully-connected means that each node in one layer connects with a certain weight w_{ij} to every node in the following layer. An MLP consists of at least three layers: an input layer, one or more hidden layers, and an output layer. This architecture is sometimes referred to as *vanilla* network when having a single hidden layer. Most of the architectures used for simple benchmarks such as MNIST experiments are MLPs with fewer than three hidden layers.

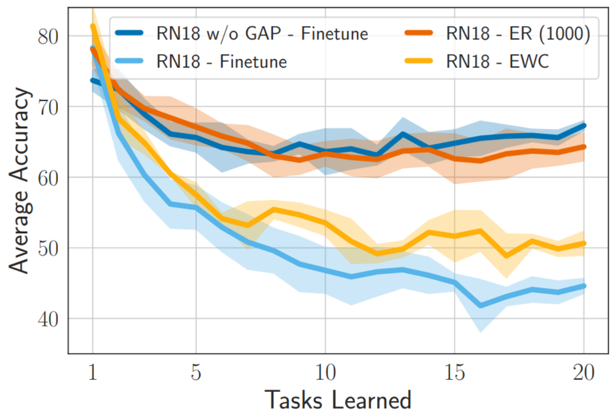


FIGURE 2.7: (Seyed-Iman Mirzadeh, Chaudhry, et al., 2022). While compared to naive fine-tuning, continual learning algorithms such as Elastic Weight Consolidation (Kirkpatrick et al., 2016) and Experience Replay (Riemer et al., 2018) improve the performance, a simple modification to the architecture such as removing the global average pooling (GAP) layer can match the performance of ER with a replay size of 1000 examples.

Convolutional Neural Networks (CNN) are required for more complex tasks of imagery. These networks are based on convolution filters that slide along input features, providing translation equivariant responses known as feature maps. Most simple CNNs, the ones considered in this section, have only convolutional layers with a stride (number of pixels that the kernel moves on each iteration) of 2 and a single fully-connected final layer for classification. These networks often have pooling layers with the objective of down-sampling in order to reduce the number of parameters.

Residual Neural Networks (ResNet) (He et al., 2015) are deep feedforward neural networks with skip connections used to jump over some layers. Typical ResNet models are implemented with double or triple layer skips with non-linearities like Rectified Linear Units (ReLU) and batch normalization (re-scaling inputs at a layer) in between.

In feedforward neural networks (where connections between nodes do not form a cycle), it is meaningful to define the concepts of *encoder* and *decoder*, even if they are defined in loose terms. An encoder is formed by the first layers of the network, it takes the non-fixed size input (although in our experiments the input size is fixed for simplification) and transforms it to a fixed-size state. This state is the input of the decoder, which is formed by the last layer(s) of the network. In most of the architectures described in the experiments, a single fully-connected classification layer is used as a decoder.

However, in a task-incremental learning scenario, the decoder can be designed to be task-specific, also known as **multi-head decoder**. This means that the classifier layer (or last few layers), which is known as head, is not shared among all tasks. There is one head per task, and the task-id indicates to the agent which head to activate with each sample.

This way, catastrophic forgetting is partially alleviated since ideally the most task-specific knowledge is coded in the heads and the common features are stored in the shared layers. The opposed concept, **single-head decoder**, is used when no task-id is provided, that is, class-incremental learning.

2.11 Optimizers

Optimizers are algorithms used to minimize the loss function, and depending on parameters like the learning rate or momentum, they *decide* the optimization step, that is, the values that the learnable parameters will have after each update. They are dependent on the model's parameters and are orthogonal to the Continual Learning method that we are using. In this section, we explain the optimizers that have been used in the experiments.

2.11.1 Vanilla Gradient Descent

Gradient descent is a way to minimize an objective function $\mathcal{L}(\boldsymbol{\theta})$ parameterized by a model's parameters $\boldsymbol{\theta}$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$ w.r.t. to the parameters (Ruder, 2016). The learning rate γ determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley. Vanilla gradient descent computes the gradient of the loss function with regard to $\boldsymbol{\theta}$ for the entire training dataset (entire experience in CL terms):

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta}_{old} - \gamma \cdot \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$$

As it is needed to calculate the gradients for a whole dataset to perform only *one* update, this optimization technique can be intractably slow. Furthermore, the lack of fluctuation in the trajectory causes the method to explore only one local valley, making it very difficult to reach better minimums.

2.11.2 Stochastic Gradient Descent

Stochastic gradient descent (SGD) in contrast to vanilla GD performs a parameter update for each mini-batch. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily, which is beneficial for *jumping* between different valleys and therefore finding a global minimum. It is the most common optimizer used in the bibliography. The pseudocode is at alg. 1.

In the pseudocode, g_t refer to the gradients at time step t, and b_t indicates the new direction of the gradient taking into account the momentum.

Procedure 1 SGD algorithm from PyTorch (*SGD PyTorch* 2019)

```
Inputs:
    $\gamma$ (learning rate), $\theta_0$ (parameters), $f(\theta)$ (objective), $\lambda$ (weight
    decay), $\mu$ (momentum)
for $t = 1 \to \dots$ do
    $g_t \leftarrow \nabla_{\boldsymbol{\theta}} f_t(\boldsymbol{\theta}_{t-1})$
    if $\lambda \neq 0$ then
        $g_t \leftarrow g_t + \lambda \boldsymbol{\theta}_{t-1}$    ▷ L2 penalty for shrinking weights and avoiding overfitting
    end if
    if $\mu \neq 0$ then    ▷ previous update direction is retained to simulate inertia
        if $t > 1$ then
            $\boldsymbol{b}_t \leftarrow \mu \boldsymbol{b}_{t-1} + g_t$
        else
            $\boldsymbol{b}_t \leftarrow g_t$
        end if
        $g_t \leftarrow \boldsymbol{b}_t$
    end if
    $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \mu g_t$
end for
Output:
    $\boldsymbol{\theta}_t$
```

2.11.3 Adam optimizer

Adam (Adaptive Moment Estimation) (Kingma and Ba, 2014) optimizer is a method that computes adaptive learning rates for each parameter. Adam is more locally stable than SGD and is less likely to converge to the minima at the flat or asymmetric basins/valleys, which often have better generalization performance over other type minima. Zhou et al., 2020 show in their results that SGD has a better generalization performance over Adam. However, Adam converges faster and Choi et al., 2019 empirically prove that the additional and expensive tuning of the three hyperparameters significantly improves Adam. It inherits properties from Adagrad (Adaptive Gradient Descent) (Duchi, Hazan, and Singer, 2011) and RMS Prop (Root Mean Square) (Hinton, 2012) optimizers, not covered in this thesis. It is computationally efficient and has little memory requirements. The pseudocode provided by the PyTorch implementation is at alg. 2.

The betas are coefficients used for computing running averages of the gradient and its square, which is intended to act similarly to the momentum.

Procedure 2 Adam algorithm from PyTorch (*Adam PyTorch* 2019)

```
Inputs:
    $\gamma$ (learning rate), $\beta_1, \beta_2$ (betas), $\theta_0$ (parameters), $f(\theta)$ (objective), $\lambda$
    (weight decay)
Initialize:
    $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1 \rightarrow \dots$ do
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    if $\lambda \neq 0$ then
        $g_t \leftarrow g_t + \lambda\theta_{t-1}$    ▷ L2 penalty for shrinking weights and avoiding overfitting
    end if
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
    $\widehat{m_t} \leftarrow m_t/(1-\beta_1^t)$
    $\widehat{v_t} \leftarrow v_t/(1-\beta_2^t)$
    $\theta_t \leftarrow \theta_{t-1} - \widehat{m_t}/(\sqrt{\widehat{v_t}} + \varepsilon)$
end for

Output:
    $\theta_t$
```

Chapter 3

Algorithms

3.1 A categorization of CL methods

In this section, different CL perspectives and multiple methods for each approach are described, compared, and discussed. Conceptually, they can be divided into (i) methods that train the network while regularizing to prevent catastrophic forgetting with previously learned tasks, (ii) methods that complement the model with a memory storing previous information, for example, by using memory replay to consolidate internal representations, and (iii) methods that selectively train the network and expand it if necessary to represent new tasks. Some methods use both a regularization term in the loss as in (i) and a small episodic memory as in (ii). They are often known as hybrid methods.

3.2 Regularization based methods

Regularization based methods are usually inspired by neuroscience findings suggesting that consolidated knowledge can be secured from forgetting through synapses. From a computational perspective, an extra parameter (apart from the weight) is associated to each weight with the importance of the corresponding weight. These parameters explain *how plastic* a connection is. These methods alleviate catastrophic forgetting by adding a regularization term in the loss function, which depends on the importance measure of each parameter. The aim of this term is to protect important weights holding previous knowledge from being modified.

Prior-focused regularization methods are the main interest within regularization-based methods. These methods are based on Bayesian inference. The model retains a distribution over the parameters that indicates the plausibility of the parameters of the model given the observed data (posterior probability distribution). When new data are available, the previous posterior distribution is combined with the likelihood of the new data given the previous model. Multiplying and normalizing these terms yields to the new posterior, and this can be repeated recursively with each new experience, where posterior becomes prior. The previous posterior constrains parameters that strongly influence prediction. The parameters are noted as $\boldsymbol{\theta}$, and $\mathcal{D}_i$ denotes the dataset for experience i.

$$\Pr(\boldsymbol{\theta}|\mathcal{D}_i, \mathcal{D}_{i-1}, \ldots, \mathcal{D}_1) = \frac{\Pr(\mathcal{D}_i|\boldsymbol{\theta}) \cdot \Pr(\boldsymbol{\theta}|\mathcal{D}_{i-1}, \ldots, \mathcal{D}_1)}{\Pr(\mathcal{D}_i)}$$

By taking the logarithms,

$$\underbrace{\log(\Pr(\boldsymbol{\theta}|\mathcal{D}_i, \mathcal{D}_{i-1}, \ldots, \mathcal{D}_1))}_{\text{posterior}} = \underbrace{\log(\Pr(\mathcal{D}_i|\boldsymbol{\theta}))}_{\text{log-likelihood}} + \underbrace{\log(\Pr(\boldsymbol{\theta}|\mathcal{D}_{i-1}, \ldots, \mathcal{D}_1))}_{\text{previous posterior (=prior)}} - \log(\Pr(\mathcal{D}_i))$$

Note that the log-likelihood $\log(\Pr(\mathcal{D}_i|\boldsymbol{\theta}))$ is the negative of the cross-entropy loss function $\mathcal{L}$ explained in section 2.6. Therefore, minimizing loss is equal to maximizing the posterior. The term on the right $\log(\Pr(\mathcal{D}_i))$ is not $\boldsymbol{\theta}$-dependent, so it is not useful for updating the model.

The left-hand side of the formula describes the posterior given the entire dataset seen so far, and the right-hand depends only on the loss function for task i. This means that all the information about the importance of the parameters for previous tasks must be captured in the previous posterior. However, this previous posterior is intractable and an approximation is required, even when forming the first posterior $\Pr(\boldsymbol{\theta}|\mathcal{D}_1) \approx q_1(\boldsymbol{\theta}) = \text{proj}(\Pr(\boldsymbol{\theta})\Pr(\mathcal{D}_1|\boldsymbol{\theta}))$. Here $q(\boldsymbol{\theta}) = \text{proj}(p^*(\boldsymbol{\theta}))$ denotes a projection operation that takes the intractable un-normalized distribution and returns a tractable normalized approximation $q(\boldsymbol{\theta})$. Having approximated the first posterior by $q_1(\boldsymbol{\theta})$, subsequent approximations can be calculated recursively by combining the previous approximate posterior distribution q_{i-1} with the likelihood $\Pr(\mathcal{D}_i|\boldsymbol{\theta})$ and projecting, that is, $\Pr(\boldsymbol{\theta}|\mathcal{D}_i, \ldots, \mathcal{D}_1) \approx q_i(\boldsymbol{\theta}) = \text{proj}(q_{i-1}(\boldsymbol{\theta})\Pr(\mathcal{D}_i|\boldsymbol{\theta}))$. Different strategies are used to find appropriate approximations. Even though some of the methods below are not originally defined from this Bayesian perspective, they all can be interpreted this way.

3.2.1 Elastic Weight Consolidation (EWC)

This work was published by Kirkpatrick et al., 2016. It was the first Continual Learning paper to approach the problem from a Bayesian perspective, and it is one of the most known papers in the field, often taken as a baseline.

The previous posterior is approximated by a Gaussian distribution with mean given by the parameters $\boldsymbol{\theta}$ after having trained on task $i-1$, and a diagonal precision given by the diagonal of the Fisher information matrix $\mathbb{F}$, $\Pr(\boldsymbol{\theta}|\mathcal{D}_{i-1}, \ldots, \mathcal{D}_1) \approx \mathcal{N}(\boldsymbol{\theta}^*_{i-1}, [\mathbb{F}]^{-1})$, where $\boldsymbol{\theta}^*_{i-1}$ indicates the optimal parameters after the completion of learning task $i-1$.

This is justified with the second order Taylor approximation of the loss function:

$$\mathcal{L}(\boldsymbol{\theta}) \approx \mathcal{L}(\boldsymbol{\theta}^*_{i-1}) + \underbrace{\left.\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}^*_{i-1}}}_{=0} + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^*_{i-1})^\top \left.\frac{\partial^2 \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}^*_{i-1}} (\boldsymbol{\theta} - \boldsymbol{\theta}^*_{i-1})$$

The first-order term is assumed to be equal to zero, as it is evaluated at the optimum parameters for task $i-1$, so it is a local minimum. If the loss function is cross-entropy loss (section A.1), then $\mathcal{L}(\boldsymbol{\theta}) = -\log \Pr(\boldsymbol{\theta}|\mathcal{D}_{i-1})$. By replacing the loss by the posterior,

$$\log \Pr(\boldsymbol{\theta}|\mathcal{D}_{i-1}) = \log \Pr(\boldsymbol{\theta}^*_{i-1}|\mathcal{D}_{i-1}) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^*_{i-1})^\top \left.\frac{\partial^2 \log \Pr(\boldsymbol{\theta}|\mathcal{D}_{i-1})}{\partial \boldsymbol{\theta}^2}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}^*_{i-1}} (\boldsymbol{\theta} - \boldsymbol{\theta}^*_{i-1})$$

Therefore, removing the logarithm we have the following result:

$$\begin{aligned}\Pr(\boldsymbol{\theta}|\mathcal{D}_{i-1}) = {} & \underbrace{\Pr(\boldsymbol{\theta}^*_{i-1}|\mathcal{D}_{i-1})}_{\text{normalizing constant}} \cdot \exp\left(\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^*_{i-1})^\top \left.\frac{\partial^2 \log \Pr(\boldsymbol{\theta}|\mathcal{D}_{i-1})}{\partial \boldsymbol{\theta}^2}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}^*_{i-1}} (\boldsymbol{\theta} - \boldsymbol{\theta}^*_{i-1})\right) \\ & \sim \mathcal{N}\left(\boldsymbol{\theta}^*_{i-1}, \left(-\left.\frac{\partial^2 \log \Pr(\boldsymbol{\theta}|\mathcal{D}_{i-1})}{\partial \boldsymbol{\theta}^2}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}^*_{i-1}}\right)^{-1}\right)\end{aligned}$$

Note that

$$\mathbb{F} := \mathbb{E}\left[\left(\frac{\partial}{\partial \boldsymbol{\theta}} \log f(X;\boldsymbol{\theta})\right)^2 \middle| \boldsymbol{\theta}\right] = -\mathbb{E}\left[\frac{\partial^2 \log f(X;\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2} \middle| \boldsymbol{\theta}\right]$$

This equality holds since

$$\frac{\partial^2}{\partial \boldsymbol{\theta}^2} \log f(X;\boldsymbol{\theta}) = \frac{\frac{\partial^2}{\partial \boldsymbol{\theta}^2} f(X;\boldsymbol{\theta})}{f(X;\boldsymbol{\theta})} - \left(\frac{\frac{\partial}{\partial \boldsymbol{\theta}} f(X;\boldsymbol{\theta})}{f(X;\boldsymbol{\theta})}\right)^2 = \frac{\frac{\partial^2}{\partial \boldsymbol{\theta}^2} f(X;\boldsymbol{\theta})}{f(X;\boldsymbol{\theta})} - \left(\frac{\partial}{\partial \boldsymbol{\theta}} \log f(X;\boldsymbol{\theta})\right)^2$$

and

$$\mathrm{E}\left[\frac{\frac{\partial^2}{\partial \boldsymbol{\theta}^2} f(X;\boldsymbol{\theta})}{f(X;\boldsymbol{\theta})} \middle| \boldsymbol{\theta}\right] = \frac{\partial^2}{\partial \boldsymbol{\theta}^2} \int f(x;\boldsymbol{\theta})\, dx = 0$$

Here, $\log f(X;\boldsymbol{\theta})$ is the logarithm of the output of the model after training on task $i-1$ and with the training samples of task $i-1$. This is calculated at the end of the learning of a task, and next task cannot begin until $\mathbb{F}$ is totally calculated. This matrix has multiple key properties. As shown, it is equivalent to the second derivative (Hessian matrix) of the loss near a minimum, and it can be computed from first-order derivatives, and therefore it is easy to calculate even for large models. Furthermore, it is positive semidefinite, which ensures that unimportant weights will never have a negative importance, so the regularization constraint is always a non-negative penalty.

Given this approximation, the loss function that is minimized in EWC is

$$\mathcal{L}(\theta) = \mathcal{L}_{\mathcal{D}_i}(\theta) + \frac{\lambda}{2}\sum_j \mathbb{F}_j \cdot (\theta_j - \theta^*_{j,\mathcal{D}_{i-1}})^2$$

where $\mathcal{L}_{\mathcal{D}_i}(\theta)$ is the loss for task i, j indicates the index of each of the parameters in θ, and $\theta^*_{j,\mathcal{D}_{i-1}}$ indicates the optimal parameters after finishing to learn task $i-1$.

3.2.2 Synaptic Intelligence (SI)

This work was published by Zenke, Poole, and Ganguli, 2017. The accuracy results are similar to EWC on the permuted MNIST benchmark. Its main algorithmic improvement with respect to EWC is that the new task importance weights are no longer computed in a separate consolidation phase after training on a task.

The loss function they use is

$$\mathcal{L}(\theta) = \mathcal{L}_{\mathcal{D}_i}(\theta) + \frac{\lambda}{2}\sum_k \Omega^i_k \cdot (\theta_k - \theta^*_{k,\mathcal{D}_{i-1}})^2$$

Here the *regularization strength* parameter is also $\frac{\lambda}{2}$, and the importance of each parameter is given by Ω^i_k.

To calculate the importance of each parameter in an *online* manner (at each training step instead of in the consolidation phase at the end of the epoch), they maintain an online importance measure ω^i_k. This measures how each parameter affects the change in loss over the trajectory in the learning phase.

Let $\theta(t)$ be the trajectory in the parameter space during training. To compute the change in loss over an entire trajectory through parameter space, we have to compute the path integral of the gradient vector field along the parameter trajectory from the initial point (at time t_{i-1}) to the final point (at time t_i).

$$\mathcal{L}(t_i) - \mathcal{L}(t_{i-1}) = \int_C \boldsymbol{g}(\boldsymbol{\theta}(t))d\boldsymbol{\theta} = \int_{t_{i-1}}^{t_i} \boldsymbol{g}(\boldsymbol{\theta}(t)) \cdot \boldsymbol{\theta}'(t)dt = \sum_k \underbrace{\int_{t_{i-1}}^{t_i} g_k(t)\theta'_k(t)dt}_{\text{parameter specific}} \equiv -\sum_k \omega^i_k$$

The first equality is due to the gradient being a conservative field, where the value of the integral is equal to the difference in loss between the end point and the start point, regardless of the trajectory taken. The ω^i_k has now the intuitive interpretation of parameter specific contribution to changes in total loss. In practice, it is computed as the product of the k-th component of the gradient $g_k = \frac{\partial \mathcal{L}}{\partial \theta_k}$ and the parameter update $\theta'_k(t)$.

The per-parameter regularization strength is computed as following:

$$\omega_k^i = \sum_{j=1}^{i} \frac{\omega_k^j}{(\theta_k(t_j) - \theta_k(t_{j-1}))^2 + \xi}$$

with ξ being a damping parameter to avoid divisions by zero.

3.2.3 Memory Aware Synapses (MAS)

This work was published by Aljundi et al., 2018. In this case, as in Synaptic Intelligence, the importance weights approximate the sensitivity of the learned function to a parameter change rather than a measure of the (inverse of) parameter uncertainty.

This approach does not depend on the ground truth labels, that is, it calculates the importance weights in an unsupervised manner. For a given data point x_j, the output of the network is $f(x_j; \theta)$. A small perturbation δ in the parameters θ results in a change in the output that using first-order Taylor approximation,

$$f(x_j : \theta + \delta) - f(x_j : \theta) \approx \sum_k g_k(x_j)\delta_k$$

where $g_k = \frac{\partial(f(x_j;\theta))}{\partial \theta_k}$ is the gradient. Assuming a small constant change δ_k, we can measure the importance of a parameter by the magnitude of the gradient g_k. The importance weight Ω_k for parameter θ_k is calculated as follows:

$$\Omega_k^i = \frac{1}{N_i} \sum_{j=1}^{N_i} ||g_k(x_j)||$$

In this way, Ω_k^i is updated in an online fashion at each new data point x_j. N_i is the number of data points available in the current task i. $|| \cdot ||$ refers to the ℓ_2-norm. The loss function used is the same as EWC and SI:

$$\mathcal{L}(\theta) = \mathcal{L}_{\mathcal{D}_i}(\theta) + \frac{\lambda}{2} \sum_k \Omega_k^i \cdot (\theta_k - \theta^*_{k,\mathcal{D}_{i-1}})^2$$

3.2.4 Kronecker factored online Laplace approximation (KFA)

This work was published by Ritter, Botev, and Barber, 2018. The approach is very similar to EWC in the sense that they calculate an approximation to the Hessian matrix after learning a task.

EWC does not take into account interactions between parameters, since only the diagonal of the matrix is calculated. The main improvement compared to EWC is that the Hessian is approximated by diagonal blocks instead of a diagonal matrix. These blocks correspond to interactions within each layer, therefore making the matrix block-diagonal. In order to reduce computations, this blocks happen to be Kronecker-factored (product of two smaller matrices). Intuitively, more importance parameters are used than in other regularization methods, so the Hessian approximation is more accurate. Furthermore, calculating the Kronecker factors requires the same calculations as a forward and backward pass through the network plus two additional matrix-matrix products. The overall cost is roughly equivalent to that of an additional training epoch compared to the other regularization methods explained so far.

In this method, we denote a NN as taking input $a_0 = x$ and producing output h_L. This input is passed through layers $1, \dots, L$ as the linear pre-activations $h_l = W_l a_{l-1}$ for $l \in 1, \dots, L$, with W_l being the weights' matrix of the layer, and $a_l = f_l(h_l)$. f_l is a non-linear elementwise function, typically a ReLU. The outputs then parametrize the log-likelihood $\log p(\mathcal{D}|h_L)$ of the data. Using the chain rule, we write the Hessian with regard to the weights of layer l as:

$$H_l = \frac{\partial^2 log p(\mathcal{D}|h_L)}{\partial \operatorname{vec}(W_l) \partial \operatorname{vec}(W_L)} = \mathcal{Q}_l \, \mathcal{H}_l$$

where H_l is the Hessian matrix of a layer l, $\operatorname{vec}(W_l)$ is the weight matrix of layer l stacked into a vector. Then, if we define $\mathcal{Q}_l = a_{l-1} a_{l-1}^\top$ and $\mathcal{H}_l = \frac{\partial^2 \log p(\mathcal{D}|\theta)}{\partial h_l \partial h_l}$.

Even if this method still makes some independence assumptions about the weights, interactions within the same layer are accounted for. The results on Permuted MNIST benchmark are better than those of EWC or SI methods.

3.3 Memory-replay based algorithms

Memory-replay based methods have a memory where they store samples in raw format or generate pseudo-samples with a generative model. The latter approach is not addressed in this work.

The stored samples from previous tasks are replayed while learning a new task to alleviate forgetting. They can be used as inputs for *rehearsal*, or to constrain fine-tuning of the new task loss to prevent previous task forgetting, similarly as in regularization-based methods.

The main drawback of replay methods that require preserving samples from all previous experiences is limited scalability, which requires additional computation and storage of raw input samples. Although fixing the memory size limits memory consumption, this also deteriorates the ability of exemplar sets from memory to represent the original distribution. Additionally, the storage of raw input samples may also cause privacy issues.

3.3.1 Incremental Classifier and Representation Learning (iCaRL)

This work was published by Rebuffi, Kolesnikov, and Lampert, 2016. It is a pure rehearsal method, which means that it explicitly retrains on a subset of stored samples in a fixed-size memory while training on new tasks.

Note that following the CL objective of efficiency (section 2.1), the memory (of size K) cannot increase while learning new tasks. If no task is privileged in terms of memory allocation, at task t, $\frac{K}{t}$ memory can be allocated to samples from each task.

iCaRL uses a *nearest-mean-of-exemplars* classification strategy. When learning a new task i, it computes the average feature vector of all exemplars for a class y. Let P_y be the subset of $\mathcal{D}_i$ such that if $(x', y') \in P_y$, then $y' = y$. $\mu_y = \frac{1}{|P_y|} \sum_{(x,y) \in P_y} \varphi(x)$. Here, $\varphi(x)$ is the feature vector of the model (typically the layer before the classifier) when having x as input. Then, the stored samples for task i are the (x, y) that have the smallest difference on their features with the average feature vector: $||\varphi(x) - \mu_y||$. Note that this method does not need a task-id, thus it is useful to apply it in class-incremental settings.

This average feature vector changes every time a learning step is made. Therefore, the memory allocation must be done dynamically. At each step of the iteration, one more example of the current training set is added to the exemplar set, namely the one that causes the average feature vector over all exemplars to best approximate the average feature vector over all training examples. Thus, the exemplar "set" is really a prioritized list. The order of its elements matters, with exemplars earlier in the list being more important. This prioritized construction was introduced by Welling, 2009 to create a representative set of samples from a distribution. In this way, when having to remove exemplars from a task, the last samples from the list are simply removed.

However, the main drawbacks are that it could overfit to the exemplar memory, it does not allow positive backward transfer, and when the number of tasks increases, each task is assigned less memory, hence the scalability over the number of tasks is very limited.

3.3.2 Gradient Episodic Memory (GEM)

This work was published by Lopez-Paz and Ranzato, 2017. It is a constrained rehearsal method, which uses the samples from memory for regularization purposes.

The key idea is to only constrain new task updates to not interfere with previous tasks. This is achieved by projecting the estimated direction of the gradient in the region outlined by previous task gradients in order to reach *positive* backward transfer.

When observing a tuple (x, y, t), the problem they solve is to find the parameters that minimize loss on current sample, subject to the fact that for every previous task, the loss on the set of samples from the memory must not increase compared to the loss of the previous model (after finishing learning previous task) on the same set. More formally,

$$\begin{aligned} \min_{\theta} \quad & \mathcal{L}(f_\theta(x,t), y) \\ \text{subject to} \quad & \mathcal{L}(f_\theta, \mathcal{M}_k) \leq \mathcal{L}(f_{\tilde{\theta}}^{t-1}, \mathcal{M}_k) \ \forall k < t, \end{aligned} \tag{3.1}$$

where $f_{\tilde{\theta}}^{t-1}$ is the predictor after learning task $t-1$.

To solve this, they used the fact that it is not necessary to store f_θ^{t-1} if the loss in previous tasks does not increase after a parameter update g. They also assume that the function is locally linear and that memory is representative enough for gradients of previous tasks:

$$\langle g, g_k \rangle := \left\langle \frac{\partial \mathcal{L}(f_\theta(x,t), y)}{\partial \theta}, \frac{\partial \mathcal{L}(f_\theta, \mathcal{M}_k)}{\partial \theta} \right\rangle \geq 0, \ \forall k < t.$$

If all $t-1$ constraints are satisfied, then the proposed g is unlikely to increase the loss at previous tasks. However, if violations occur, the projection of g to the closest in ℓ_2 norm is calculated by solving

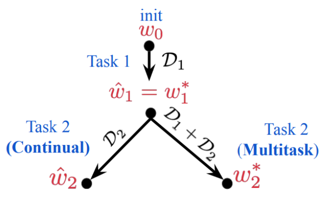
$$\begin{aligned} \min_{\tilde{g}} \quad & ||g - \tilde{g}||_2^2 \\ \text{subject to} \quad & \langle \tilde{g}, g_k \rangle \geq 0, \ \forall k < t. \end{aligned} \tag{3.2}$$

The projection is made on a gradient, which has the dimension of the number of parameters of the model, making it intractable for finding a minimum. There are only $t-1$ restrictions; therefore, by formulating the dual of the Quadratic Program, it can be solved on only $t-1$ variables.

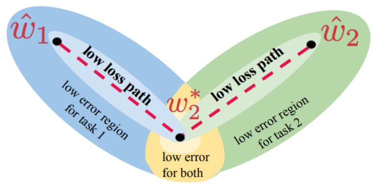
3.3.3 Averaged Gradient Episodic Memory (A-GEM)

This work was published by Chaudhry et al., 2018, heavily inspired in GEM method (subsection 3.3.2). It is an improved version of GEM, which enjoys the similar performance as GEM, while being almost as computationally and memory efficient as EWC (subsection 3.2.1) and other regularization-based methods.

GEM method is hardly scalable. In fact, at each training step, GEM computes all the gradients $g_k, \forall k < t$ using all samples from the episodic memory, and it also needs to solve the Quadratic Program. This becomes prohibitive when the size of $\mathcal{M}$ and the number of tasks is large. To alleviate this computational burden, Chaudhry et al., 2018 propose Averaged GEM. In this case, as opposed to Equation 3.1, there is only one *average* episodic memory:



(A) First, $\hat{w}_1$ is learned on task 1. Then, we either reach $\hat{w}_2$ by fine-tuning second task or w_2^* by training jointly on both tasks.



(B) Linear connectivity between w_2^* and $\hat{w}_1$ and between w_2^* and $\hat{w}_2$

FIGURE 3.1: (Seyed-Iman Mirzadeh, Farajtabar, Görür, et al., 2020). While offline learning learns in a single phase from a static set of data, Continual Learning systems learn from a stream of non-stationary data.

$$\begin{aligned} \min_{\theta} \quad & \mathcal{L}(f_\theta(x,t),y) \\ \text{subject to} \quad & \mathcal{L}(f_\theta,\mathcal{M}) \le \mathcal{L}(f_{\tilde{\theta}}^{t-1},\mathcal{M}) \quad \text{where } \mathcal{M} = \bigcup_{k<t} \mathcal{M}_k, \end{aligned}$$

The optimization problem in Equation 3.2 now has only one constraint,

$$\begin{aligned} \min_{\tilde{g}} \quad & ||g-\tilde{g}||_2^2 \\ \text{subject to} \quad & \langle \tilde{g}, g_{ref} \rangle \ge 0, \end{aligned}$$

where g_{ref} is a gradient computed using a a batch randomly sampled from the episodic memory of all the past tasks. Therefore, this constrained optimization problem can now be solved very quickly. When g violates the constraint, it is projected as:

$$\tilde{g} = g - \frac{\langle g, g_{ref} \rangle}{\langle g_{ref}, g_{ref} \rangle} g_{ref}$$

This improves GEM in the following ways: computationally, fewer gradients are calculated and it saves the calculation of the QP. Furthermore, less violations will happen as there is only one constraint instead of the number of past tasks. This will be particularly noticeable when the number of tasks is large.

3.3.4 Linear Mode Connectivity

This work was published by Seyed-Iman Mirzadeh, Farajtabar, Görür, et al., 2020. It was motivated by the question *what property does multitask or joint learning have that a continual algorithm does not? Can we reverse engineering this process? Multitask learning (section 2.8) is claimed to be an upper bound for continual learning, so these questions arise naturally when trying to come closer to this bound.*

After learning the first task, w_1^* is learned. Then, if naive training (fine-tuning) is used for learning task 2, the learned weights are noted as $\hat{w}_2$. However, if the model is re-trained on both tasks (joint or multitask training), we refer as w_2^* to the optimal weights. The goal is to compare w_2^* and $\hat{w}_2$ to find any relationship between these minimums. They reach the conclusion that comparing Euclidean Distance in the parameters does not provide relevant information in this case, and neither does comparing representations (on the latent layer) through Centered Kernel Alignment (Kornblith et al., 2019).

The need to find another metric to compare these weights inspires the mode connectivity concept. The hypothesis is that optima of loss functions are connected by simple curves over which training and test accuracy are nearly constant (low loss over the curve). Linear mode connectivity is the particular case where these curves are linear. Seyed-Iman Mirzadeh, Farajtabar, Görür, et al., 2020 show that the multitask solution to tasks 1 and 2 is linearly connected with continual solutions, but $\hat{w}_1$ and $\hat{w}_2$ are not (Figure 3.1b, from Seyed-Iman Mirzadeh, Farajtabar, Görür, et al., 2020).

The idea of the method is to regularize the loss function to find minima that are linearly connected to previous minima, s.t. from $\hat{w}_1$ we can move to w_2^* instead of moving to $\hat{w}_2$. Therefore, the proposed minimization problem is

$$\bar{w} = \operatorname{argmin}_w \int_{0\leq\alpha\leq 1} [\underbrace{\mathcal{L}_1(\hat{w}_1 + \alpha(w - \hat{w}_1))}_{\text{connected to task 1 solution}} + \underbrace{\mathcal{L}_2(\hat{w}_2 + \alpha(w - \hat{w}_2))}_{\text{connected to task 2 solution}}] d\alpha$$

This equation is approximated in the following way when implemented (for task t):

$$\bar{w}_t = \operatorname{argmin}_w \sum_{\alpha} [\mathcal{L}_{t-1}(\bar{w_{t-1}} + \alpha(w - \bar{w_{t-1}})) + \mathcal{L}_t(\hat{w}_t + \alpha(w - \hat{w}_t))] d\alpha$$

Here, α is sampled uniformly from $[0, 1]$. Note that following Continual Learning desiderata section 2.1, there should be no access to previous experiences. Thus, a memory is needed for connecting the solutions of the previous and the current task. This way, even if it is a replay method, it only stores samples from task $t-1$, needed for the calculation of $\mathcal{L}_{t-1}$. Consequently, there are no memory problems when increasing number of tasks.

3.3.5 Subspace Connectivity

Although the Subspace Connectivity algorithm (Doan et al., 2022) is not an *ensemble method*, it is heavily motivated by this concept. *Ensemble methods* in artificial neural networks combine the predictions made by independently trained multiple base models in order to produce an optimal predictive model.

Doan et al., 2022 adapt the concept of *ensemble models* to Continual Learning scenarios. Given n independent models, a set of weights $\{w_i\}_{i=1}^n$, a task-id t, and a data batch

(x, y), the total loss aimed to optimize is $\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_t(f_{w_i}(x), y)$, where $\mathcal{L}_t$ is the total loss after learning task t, and f_{w_i} is the predictor of the i-th model. This is the average between the losses of every model, and the average prediction is used as the output of the model, that is, $\frac{1}{n}\sum_{i=1}^{n} f_{w_i}(\cdot)$. Therefore, the training differs only in the initialization of the weights of each model. This implies that the training cost increases linearly with the number of models n.

The proposed way to reduce this training cost is through subspace models, as in Wortsman et al., 2021. Here, the number of models is still n, and the set of learnable parameters is $\{\boldsymbol{w}_i\}_{i=1}^{n}$. Learning a subspace of dimension n consists in training the predictor $f_{\bar{\boldsymbol{w}}}$ with $\bar{\boldsymbol{w}} = \sum_{i=1}^{n}\alpha_i \boldsymbol{w}_i$, $\boldsymbol{\alpha} \in \Delta^n$. In Doan et al., 2022 the $\boldsymbol{\alpha}$'s are sampled randomly following a uniform distribution over the simplex, $\boldsymbol{\alpha} \sim \mathcal{U}(\Delta^n)$.

Unlike the ensemble method, subspace methods have a slightly similar computation cost as single models. Backpropagation is made only with respect to $\bar{\boldsymbol{w}}$ (instead to all the $\boldsymbol{w}_i$'s as in the ensemble method), and the update with regard to each w_i is, assuming a standard Stochastic Gradient Descent optimizer,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_i} = \frac{\partial \mathcal{L}}{\partial \bar{\boldsymbol{w}}} \cdot \frac{\partial \bar{\boldsymbol{w}}}{\partial \boldsymbol{w}_i} = \alpha_i \cdot \frac{\partial \mathcal{L}}{\partial \bar{\boldsymbol{w}}}, \quad \forall i \in \{1, \ldots, n\}$$

However, by this subspace connectivity method, forgetting still occurs. This is because even if a flat low-loss region is found throughout the learning experience, there is no connectivity between the solutions at the end of each experience. As the subspace method is motivated by the mode connectivity, the proposed approach by Doan et al., 2022 is to connect the subspaces throughout the learning.

First, the algorithm learns a subspace solution for the incoming task t. The solution $\widehat{W}_t = \{\hat{w}_{t,i}\}_{i=1}^{n}$ fine-tuning is obtained by optimizing

$$\widehat{W}_t = \{\hat{\boldsymbol{w}}_{t,i}\}_{i=1}^{n} = \underset{W}{\operatorname{argmin}} = \mathbb{E}_{\boldsymbol{\alpha}\sim\mathcal{U}(\Delta^n)}[\mathcal{L}_t(W^T\boldsymbol{\alpha})]$$

Then, $m_{\mathcal{B}}$ samples are stored per task in a buffer memory $\mathcal{B}$. These samples are used to connect linearly solutions of two subspaces in the following way:

As the midpoint of the simplex gives the best performance (explained in section 5.1), $\hat{w}^*_{t,\text{mid}} = \frac{1}{n}\sum_{i=1}^{n}\hat{w}_{t,i}$ is taken as the most performing solution of $\widehat{W}_t$. Then, $\hat{w}^*_{t,\text{mid}}$ and $\hat{w}^*_{t-1,\text{mid}}$ are connected via a low-loss path. The loss over the connecting path acts as a regularizer term in the following way:

$$\{\boldsymbol{w}^*_{t,i}\}_{i=1}^{n} = \underset{W}{\operatorname{argmin}}\, \mathbb{E}_{\tilde{\boldsymbol{\alpha}}\sim\mathcal{U}(\Delta^{n+1})}[\mathcal{L}_t(W^T\boldsymbol{\alpha} + \alpha_{n+1}\hat{\boldsymbol{w}}_{t,\text{mid}}) + \sum_{j=1}^{t-1}\mathcal{L}_j(W^T\boldsymbol{\alpha} + \alpha_{n+1}\boldsymbol{w}^*_{t-1,\text{mid}})]$$

where $\boldsymbol{\alpha} \in \mathbb{R}^n$ and $\bar{\boldsymbol{\alpha}} = (\boldsymbol{\alpha}, \alpha_{n+1}) \in \mathbb{R}^{n+1}$, and where $\mathcal{L}_j$ is the loss on task j calculated using the samples from the buffer $\mathcal{B}$. The idea of this regularization technique is to create a low-loss linear path between the subsequent solutions $\boldsymbol{w}^*_{t-1,\text{mid}}$ and $\boldsymbol{w}^*_{t,\text{mid}}$.

An intuitive justification for this approach is that the subspace method binds the models within a single task, while the mode connectivity regularization binds the subspaces together. In order to delve more into the properties of this method, some experiments are proposed at chapter 5.

3.4 Parameter isolation based algorithms

This approach dedicates different model parameters to each task in order to prevent forgetting (Mallya and Lazebnik, 2017; Serrà et al., 2018; Rusu et al., 2016; Ke, B. Liu, and Huang, 2021). It could be seen as taking regularization-based methods to the extreme.

When architecture size is not constrained, it is possible to add new branches to the network to solve new tasks and freeze previous task parameters.

If the architecture remains static, the neural network is divided into different fixed parts that are allocated to each task. Therefore, they usually require a task oracle, so the only possible scenario is task-incremental, possibly in a multi-head setup. In fact, multi-head architectures are inherently parameter isolation experiments, even if the main algorithm is not categorized as it.

This category can be seen from the regularization perspective as adding infinite penalty to some parameters for some tasks. This design choice potentially results in no deterioration of performance, but inherently limits the amount of tasks that can be fitted in a single network. Furthermore, the interaction among past and new tasks is limited or non-existent and, as such, the conditions where catastrophic forgetting occurs are not illuminated. For this reason, the thesis does not focus on such algorithms.

Chapter 4

Continual Learning Framework

4.1 Overview

One of the main objectives of the thesis has been to reproduce the results of the most influential papers on Continual Learning. For this purpose, a CL framework has been designed in order to implement these papers and to be able to easily run different experiments choosing the settings (architecture, benchmark, CL method, optimizer, hyperparameters, etc.). *à la carte*, that is, with freedom to try multiple configurations. This framework aims to separate the engineering and research code to help both the development and usability of the framework. For example, algorithmic logic should not occur in the same place as parsing of arguments or logging. The architecture should also follow the *open-closed principle* of object-oriented programming: software entities (classes, algorithms, functions, etc.) should be open for extension but closed for modification (Meyer, 1997). For instance, an algorithm can allow building upon it without modifying its source code and its internal abstractions.

Various full-featured libraries with this purpose already exist in this field. They facilitate both research and development, as having all algorithms written with different frameworks over isolated repositories is a current challenge that hinders research. The most well known libraries are *avalanche* (Lomonaco, Pellegrini, et al., 2021), *CL-Gym* (S. Mirzadeh and H. Ghasemzadeh, 2021), and *Sequoia* (Normandin et al., 2021). However, in order to deeply understand Continual Learning algorithms, methods, and implementations, we considered it more appropriate to implement our own solver without using any library. Nevertheless, the framework is inspired by these libraries, which are described below.

CL-Gym mostly focuses on the supervised continual learning (2.4.1), and our CL framework is heavily inspired in the design and components of this library. **Avalanche** is the most known library and it is very similar to CL-Gym. They share most of the components with CL-Gym and with this framework, but the main differences are that they implement evaluation and logging as main components, and their benchmarks are more computer vision oriented instead of toy-datasets as in here. **Sequoia** focuses on reinforcement learning methods for CL (2.4.3), and even if it supports some supervised learning methods, it mixes research and engineering code in a single place, which increases the complexity and clarity of the code.

4.2 Components

In this section, the components and mechanisms of the framework are explained. CL-framework contains four main components: `algorithms`, `backbones`, `benchmarks` and `callbacks`.

4.2.1 Benchmarks

The `benchmark` component includes some standard continual learning benchmarks (2.9). Currently, the supported benchmarks are Split MNIST, Rotated MNIST, Permuted MNIST, Split CIFAR-10, Split CIFAR-100 and Split TinyImageNet. They are all vision benchmarks with different difficulties.

All benchmarks must inherit the base `Benchmark` class, which contains the following methods:

- `prepare_datasets`: loads in memory the dataset, following the customized logic of each benchmark. Each experience should have its own train and test datasets.
- `load_memory (task)`: This applies only to methods requiring episodic memory. The method provides episodic memory loaders.
- `load_joint (task)`: it provides data loaders for joint training 2.8. For task t, all data from task 1 to t is loaded together.
- `load_memory_joint (task)`: it loads the episodic memory for all previous tasks instead of a single task.
- `load_augmented_with_memory (task)`: it returns a data loader containing both the memory samples from tasks 1 to t and the current experience samples. It is useful for rehearsal methods such as iCaRL (3.3.1).

If necessary, each benchmark should implement these methods. In addition to the methods, the `benchmark` component handles all the logic related to the necessary transforms in the datasets.

4.2.2 Backbones

The `Backbone` component refers to neural network architectures (2.10). The supported encoders are Convolutional Neural Networks (CNN), MultiLayer Perceptrons (MLP), and Residual Neural Networks. The choice of these backbones is due to their popularity in the bibliography. Easy-to-solve toy benchmarks, such as MNIST variants, are usually used with MLPs, while more difficult tasks require the more complex encoders. The supported classifiers are both the single-head and multi-head classifiers (supporting task-incremental, and task-incremental scenarios). The `task_router` module handles the activations of the different heads depending on the labels.

`Backbone` inherits from the PyTorch `nn.Module` class and therefore supports all the features it provides. The algorithm module has complete access to the backbone and can manipulate gradients.

4.2.3 Algorithms

The `Algorithms` component is the most important element of the framework. It is responsible for the majority of the research code. The following diagram schematically describes the `run` method of the `BaseAlgorithm`:

$$\texttt{run}\begin{cases}\texttt{run\_setup} & \begin{cases}\texttt{on\_before\_setup}\\ \texttt{on\_before\_setup\_cb}\\ \texttt{setup}\\ \texttt{on\_after\_setup}\\ \texttt{on\_after\_setup\_cb}\end{cases}\\ \texttt{run\_fit} & \begin{cases}\texttt{on\_before\_fit}\\ \texttt{on\_before\_fit\_cb}\\ \texttt{fit}\xrightarrow{\text{for each task}}\begin{cases}\texttt{on\_before\_train\_task}\\ \texttt{on\_before\_train\_task\_cb}\\ \texttt{train\_task}\xrightarrow{\text{for each epoch}}\\ \texttt{on\_after\_train\_task}\\ \texttt{on\_after\_train\_task\_cb}\end{cases}\\ \texttt{on\_after\_fit}\\ \texttt{on\_after\_fit\_cb}\end{cases}\\ \texttt{run\_teardown} & \begin{cases}\texttt{on\_before\_teardown}\\ \texttt{on\_before\_teardown\_cb}\\ \texttt{teardown}\\ \texttt{on\_after\_teardown}\\ \texttt{on\_after\_teardown\_cb}\end{cases}\end{cases}$$

$$4.2.3 \xrightarrow{\text{for each epoch}} \begin{cases} \texttt{on\_before\_train\_epoch} \\ \texttt{on\_before\_train\_epoch\_cb} \\ \texttt{train\_epoch} \xrightarrow{\text{for each batch}} \begin{cases} \texttt{on\_before\_train\_step} \\ \texttt{on\_before\_train\_step\_cb} \\ \texttt{train\_step} \\ \texttt{on\_after\_train\_step} \\ \texttt{on\_after\_train\_step\_cb} \end{cases} \\ \texttt{on\_after\_train\_epoch} \\ \texttt{on\_after\_train\_epoch\_cb} \\ \\ \texttt{on\_before\_val\_all\_tasks} \\ \texttt{on\_before\_val\_all\_tasks\_cb} \\ \texttt{val\_all\_tasks} \\ \texttt{on\_after\_val\_all\_tasks} \\ \texttt{on\_after\_val\_all\_tasks\_cb} \end{cases}$$

Note that the diagram is nonexhaustive w.r.t. all the functions of the base algorithm, as its purpose is to provide intuition on the structure of the code. In the diagram, the method `run` contains `run_setup`, `run_fit` and `run_teardown`, which are executed sequentially. Each of them have their respectives `on_before`, `on_before_cb`, `on_after`, `on_after_cb` routines. In the base algorithm most of these functions are empty and only constitute an skeleton such that the CL algorithms can inherit them according to the requirements of each algorithm. The purpose of having such a fine-grained set of functions is that as many CL methods as possible can be implemented following the *open-closed principle*. `fit` function is almost the only one with code, and it holds all the logic related to the training and validation, that is, data-loaders initialization, calling the configuration routine of the optimizer at each training task, forward and backward passes, and calling the `compute_loss` function.

The `cb` stands for `callbacks`, which are sets of functions to be applied at given stages of the training procedure. An algorithm can have multiple callbacks, and the intention of having both the *on_before and on_after* and the *on_before_cb and on_after_cb* is that all the code to be added into a method that is not related to the algorithm itself should go in the callbacks. However, the code strictly related to the algorithm should inherit the *on_before and on_after* functions. For instance, the ω_k^i parameters of the SI algorithm (3.2.2) are calculated `on_after_training_step`, and the Fisher diagonal of the EWC method (3.2.1) is `on_after_training_task`.

Algorithm	**Details**
Naive training (finetuning)	Baseline, upper bound for forgetting (2.8)
Joint training (multitask)	Baseline, upper bound for accuracy (2.8)
Elastic Weight Consolidation (EWC)	Regularization (3.2.1)
Synaptic Intelligence (SI)	Regularization (3.2.2)
Memory Aware Synapses (MAS)	Regularization (3.2.3)
Kronecker Factored Laplace Approximation (KFA)	Regularization (3.2.4)
Incremental Classifier and Representation Learning (iCaRL)	Memory-replay, Rehearsal (3.3.1)
Averaged Gradient Episodic Memory (A-GEM)	Memory-replay, Gradient projection (3.3.3)
Mode Connectivity - Stochastic Gradient Descent (MC-SGD)	Memory-replay + Regularization (Hybrid method) (3.3.4)
Subspace Connectivity	Memory-replay + Regularization (Hybrid method) (3.3.5)

TABLE 4.1: Supported algorithms

The algorithms currently supported by the framework are listed in Table 4.1.

4.3 Callbacks

This section overviews the currently implemented callbacks of the framework: `MetricCallback` and `MemoryCallback`.

`MetricCallback` computes, updates and stores the metrics (2.7) of the experiment. It also reports them to the *weights & biases* (`https://wandb.ai/`) platform. It is an experiment tracking tool for machine learning. It facilitates to keep track of experiments, compare different runs, visualize metrics and results, and share them.

`MemoryCallback` is an alternative implementation of the memory mechanism described in the methods of the benchmarks at 4.2.1.

Chapter 5

Experiments

Having the objective of understanding more deeply both the methods reviewed during the previous chapters and the properties of continual learning in general, some experiments have been developed. Experiment in section 5.1 explores a more general alternative to the Subspace Connectivity method (subsection 3.3.5). section 5.2 compares different architectures and methods and studies the importance of the width of the neural network in the performance of a CL solution. section 5.3 makes an observation on multiple benchmarks and compares the nature of forgetting with different settings and conditions. Finally, experiment section 5.4, which is heavily inspired in Seyed-Iman Mirzadeh, Farajtabar, and Hassan Ghasemzadeh, 2020, analyzes how with a wisely chosen set of hyperparameters and architectures, adding a dropout probability to the NN can be as effective as a regularization method in preventing catastrophic forgetting.

Some figures in the following experiments show only the average accuracy of the first five tasks, although being trained on 10 or 20 tasks. The reason for this is that *plasticity* is always maintained throughout the learning process, and forgetting can be mostly appreciated on the tasks that the agent has not seen in a long time. Therefore, even if the graphs might seem incomplete for this reason, the newer tasks do not provide much information and are not shown in order to avoid overly cluttered figures.

5.1 Improving Subspace Connectivity using the Dirichlet distribution

In Doan et al., 2022, detailed at subsection 3.3.5, they claim that "although one can learn a distribution over the α's, we consider the standard case as in Wortsman et al., 2021 where we sample it uniformly in the simplex Δ^n". The idea behind sampling uniformly is that there is not any region where we could lose connectivity around the simplex because of having less probability.

They also reached the conclusion that the subspace midpoint gets the best accuracy.

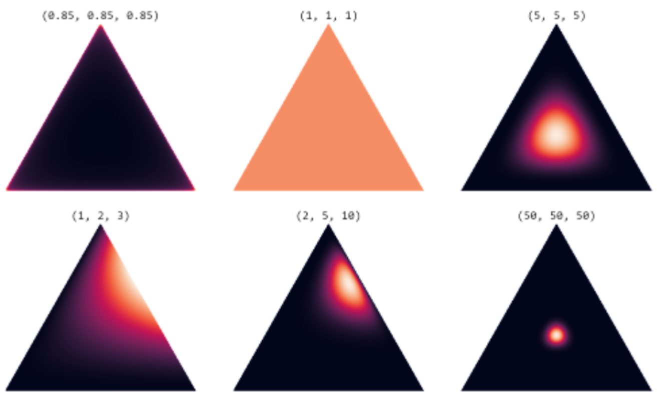


FIGURE 5.1: Image from S. Liu, 2019. Dirichlet distribution for $n = 3$ (two-simplex or equilateral triangle) for different values of β.

Therefore, it is not clear whether the uniform distribution over alphas is the best for sampling $\bar{w}$, as it might be more beneficial not to give equal importance to the center and the vertices. As the Dirichlet distribution, also known as multivariate beta distribution, is defined over an $n-1$-simplex, and the uniform distribution is a special case of it where all the parameters are equal to 1, it is an appropriate distribution for our experiment.

Dirichlet distribution, $\mathrm{Dir}(\beta)$, follows this probability density function:

$$f(\alpha_1, \ldots, \alpha_n; \beta_1, \ldots, \beta_n) = \frac{1}{\mathrm{B}(\beta) \prod_{i=1}^{n} \alpha_i^{\beta_i - 1}}$$

where $\{\alpha_i\}_{i=1}^n \in \Delta^n$, or in other words, $\sum_{i=1}^n \alpha_i = 1$, and $\alpha_i \geq 0 \forall i \in \{1, \ldots, n\}$. The normalizing constant is the multivariate beta function, which can be expressed in terms of the gamma function:

$$\mathrm{B}(\beta) = \frac{\prod_{i=1}^n \Gamma(\beta_i)}{\Gamma(\sum_{i=1}^n \beta_i)}$$

All the vertices in the simplex have the same meaning since they are just different initializations of the weights. Hence, it makes no sense to prioritize one vertex over others. We will force the distribution to give the same importance to all vertices, which means having $\beta = (\beta_1, \ldots, \beta_1)$. So, the mean is $\mathbb{E}[\alpha_i] = \frac{1}{n}$, and the distribution has one single parameter. The covariance, which in this case is equal to the variance, is $\mathrm{cov}[\alpha_i, \alpha_j] = \frac{n-1}{n^3\beta + n^2}$. If $\beta_1 > 1$, the variance decreases (more probability in the center), if $\beta_1 < 1$ the vertices have higher probability than the center, and if $\beta_1 = 1$ we have the uniform distribution over the simplex. This can be visualized in Figure 5.1 (S. Liu, 2019).

The objective of the experiment is to find a value for β that leads to a better performance of the Subspace Connectivity method than with $\beta = 1$ used in the paper. As it is an optimization problem on $\mathbb{R}^+$. We know that for $\beta \to 0$, the probability distribution

β	Avg. accuracy
limit to 0	61.8
0.5	58.3
1 (uniform)	63.8
2	67.6
5	71.0
10	69.4
20	69.6
limit to ∞ (always midpoint as in evaluation)	68.3

TABLE 5.1: Caption

becomes discrete and uniform in the vertices, and for $\beta \to \infty$, $\Pr(\alpha_i = \frac{1}{n}) = 1 \forall i \in \{1, \dots, n\}$. By doing a grid-search, the found accuracies on iCIFAR-100 benchmark are shown in Table 5.1. The experiment was made with three models on a Res-Net18 and training for 50 epochs per task.

As a possible future improvement and following the suggestion from Doan et al., 2022, this distribution could be learned over time and therefore have a β that is continually adapting.

5.2 Importance of layer width

As introduced in section 2.10, Continual Learning strategies usually need wider networks than in other ML paradigms. Therefore, it is interesting to find methods that perform well in networks that are not very wide. The goal of this experiment is to compare how width affects performance of different methods and scenarios and to draw conclusions about which settings are best suited for avoiding catastrophic forgetting on narrow networks.

5.2.1 Experiment on Split MNIST

The architecture chosen for this experiment is the one most papers use for MNIST-based benchmarks. It is a multilayer perceptron with two fully-connected hidden layers, and a multi-headed architecture, where task-ids are provided both in training and validation steps. The goal is to compare the behavior of multiple methods w.r.t. the change of the width.

In Table 5.2.1, the accuracy of the model depending on the width of the backbone is shown in the Split MNIST dataset for different methods. With 40 or more neurons per layer, the network performs well for the different methods, but for fewer neurons the capacity to accurately solve the continual learning problem drops. We would like to study the reason for this drop, under what conditions and in what way it occurs. For

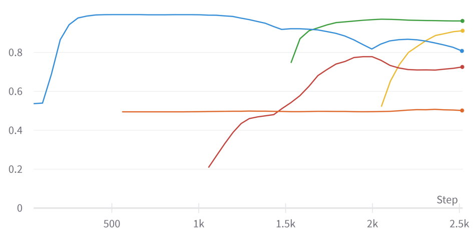
width	# params Split MNIST	# params Perm. MNIST
5	4,806	5,046
10	8,866	9,306
20	17,136	17,976
40	34,276	35,916
60	52,216	54,656
100	90,496	94,536
150	142,846	148,886
250	262,546	272,586
400	479,596	495,636

TABLE 5.2: An important observation is that in this experiment, the input layer is 28×28, the hidden layers have w neurons each and each of the 5 heads of the classifier have $|\mathcal{Y}_t|$ outputs. Therefore, the number of parameters (taking into account the bias) is $(28 \cdot 28 + 1) \cdot (w + 1) + (w + 1) \cdot (w + 1) + 5 \cdot (w + 1) \cdot |\mathcal{Y}_t| \in \mathcal{O}(w^2)$. The number of parameters grows quadratically in the number of neurons per layer. However, most of the neurons are at the first fully-connected layer, so in practice for $w \leq 400$, the growth is almost linear.

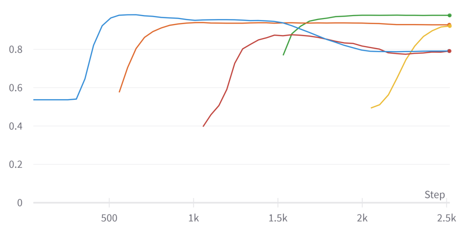
5, 10, and 20 neurons, Table 5.2.1 shows the average of multiple runs with different random initializations of the weights, as it differs a lot between different runs. For $\geq$ 40 neurons, the accuracy is stable.

Note that multi-task or joint-training has been used as a baseline (black line on Table 5.2.1), and accuracy does not drop for this non continual learning approach, even if the number of units per hidden layer is only 5.

Table 5.2 shows the number of parameters of the network for different layer widths. Following the notation of the table, $|\mathcal{Y}_t| = 2$ for this Split MNIST experiment.

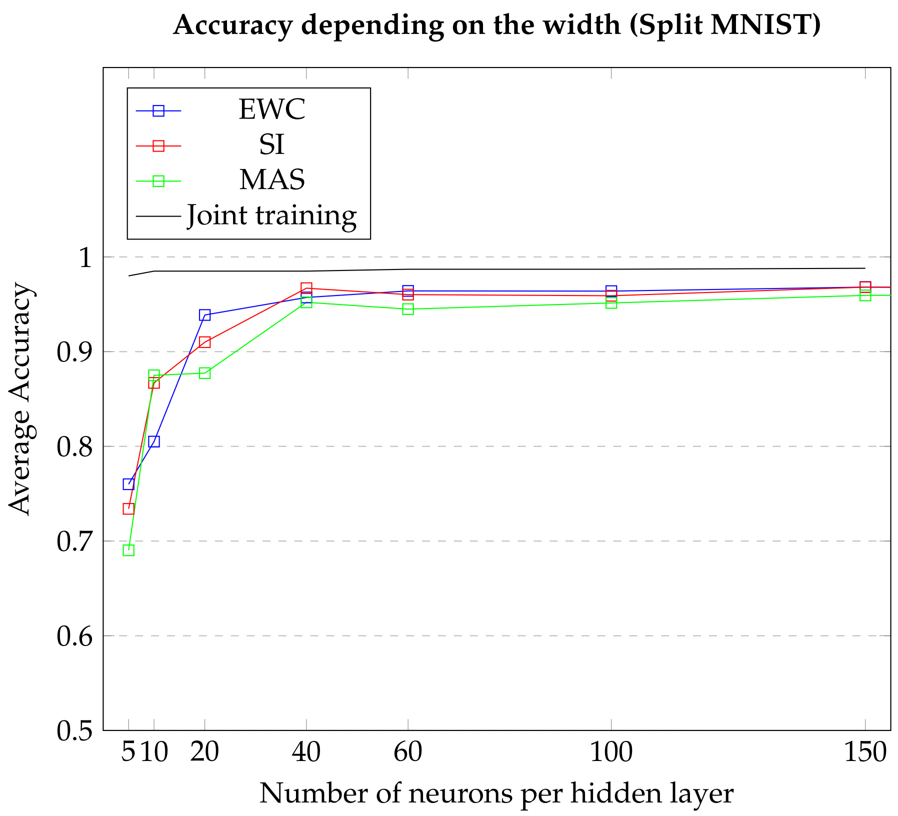


(A) Task 3 is not learned



(B) Good accuracy in all tasks

FIGURE 5.2: Two different executions of the same experiment with EWC on Split MNIST. The backbone is an MLP with two hidden layers of 10 neurons each



In Figure 5.2, two runtimes with exactly the same setup are shown. In Figure 5.2a, the average accuracy at the end of the training is worse than in Figure 5.2b, where all tasks are learned. This means that the learning is initialization-dependent for *backbones* (architectures) without enough width. However, Split MNIST might be a too simple benchmark for accurately measuring which algorithm is the least width-dependent.

5.2.2 Experiment on Permuted MNIST

A similar experiment is conducted with the Permuted MNIST benchmark for 5 different permutations, using the same architecture. This benchmark is more difficult to solve than Split MNIST, and it is expected to need a slightly bigger network to solve the problem. For a permuted MNIST experiment with 5 tasks, also with a multi-head

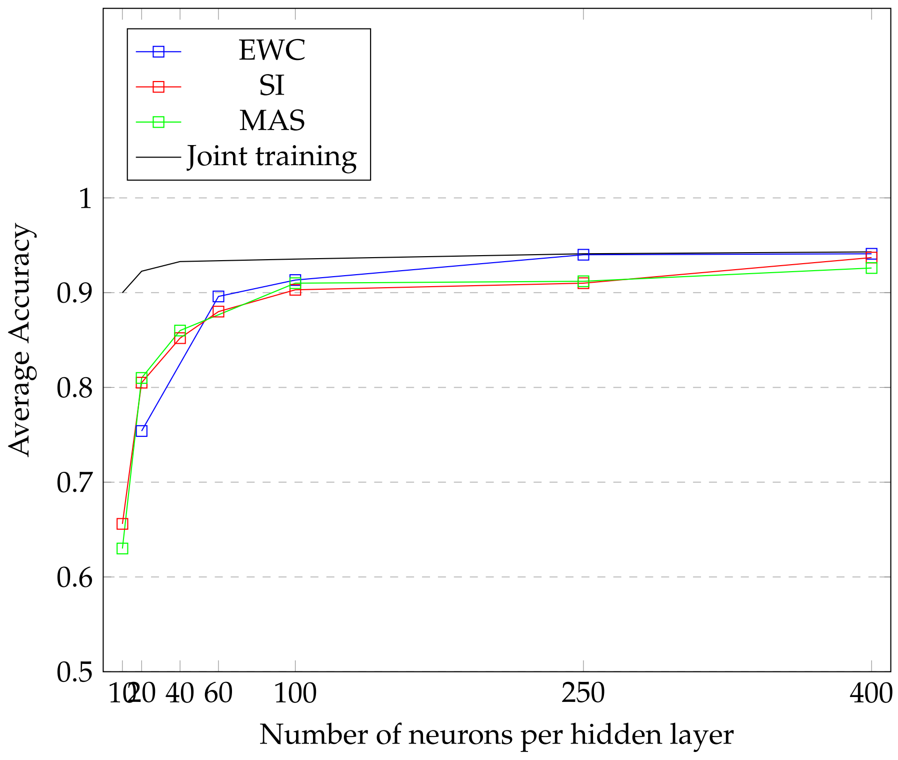


FIGURE 5.3: Accuracy depending on the width (Permuted MNIST)

setting, the number of parameters is bigger than on Split MNIST due to the classifiers at each head ($|\mathcal{Y}_t| = 10$ for every task t) (Table 5.2). The results are shown in Figure 5.3.

In this case, joint training has also a minimal drop of accuracy in narrow networks. This confirms the hypothesis of the need of wide networks for regularization methods is true. Also, the results on the three chosen classical regularization methods are very similar, and the beginning of the drop is on approximately at 60 units.

5.2.3 Experiment on Split CIFAR-100 with CNNs

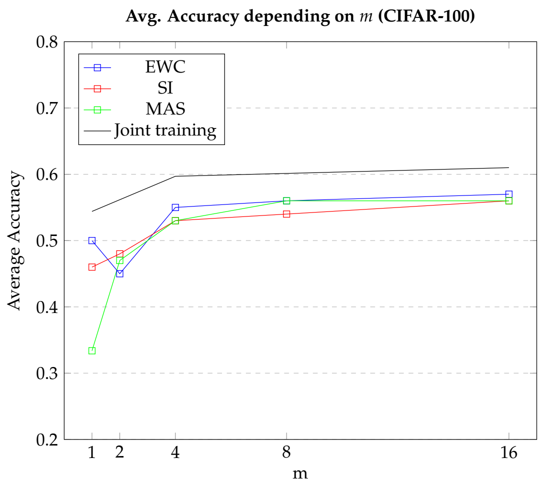
As explained in section 2.9, Split CIFAR-100 is a much more complex benchmark that requires the use of more sophisticated networks. Therefore, it is a suitable benchmark to test how the width of the network affects convolutional neural networks (section 2.10).

The chosen variation of Split CIFAR-100 splits the CIFAR-100 dataset (without adding CIFAR-10) into 20 disjoint datasets of 5 tasks each. The number of tasks is 10, so in total the algorithm must learn 50 classes.

The backbone encoder is a CNN with 3 layers, having $\{4, 8, 16\} \cdot m$ channels, where m is a multiplier to make the network wider. The kernel size of the convolution filters is 3, as the images in this dataset are relatively small. The activation function is the rectified linear unit (ReLU), and the stride is 2. Table 5.3 shows the number of parameters of the model for each multiplier.

	$m = 1$	$m = 2$	$m = 4$	$m = 8$	$m = 16$
Encoder first layer	112	224	448	896	1,792
Encoder second layer	296	1,168	4,640	18,496	73,856
Encoder third layer	1,168	4,640	18,496	73,856	295,168
Decoder	1,285	2,565	5,125	10,245	20,485
Total	2,861	8,597	28,709	103,493	391,301

TABLE 5.3: Number of parameters of the CNN depending on the multiplier m



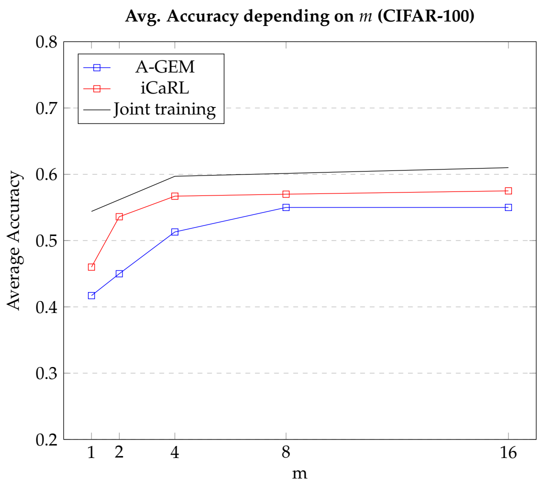


FIGURE 5.4: Experiments on regularization-based (left) and memory-replay-based (right) methods. m (there are m · [4, 8, 16] channels)

The results are shown in Figure 5.4 on the left.

Figure 5.4 on the right shows the results with the same settings in memory replay-based methods. One might expect to have better results than with regularization based methods. However, in this experiment regularization methods do not have almost any forgetting, as shown in Figure 5.7. Therefore, it makes sense that other methods that focus on having good *stability* such as A-GEM or iCaRL do not improve over the others in this case.

5.3 *Gradual* forgetting on regularization methods

Regularization methods have been shown to be a useful way to avoid forgetting. However, we would like to study the way a network behaves when it does not have enough capacity to store the information for all the tasks. This experiment intends to find how regularization-based methods avoid forgetting in comparison to naive fine-tuning on the same architecture. The results show that a regularization based method, in particular Synaptic Intelligence (the results were very similar with other regularization-based

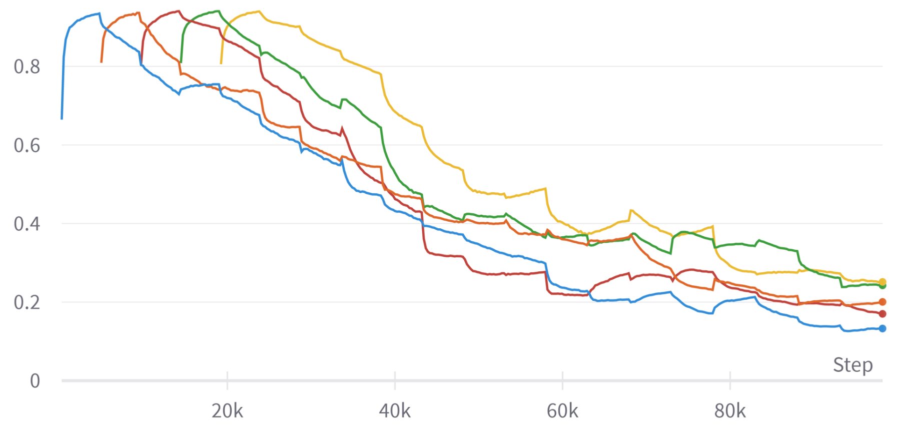


FIGURE 5.5: A regularization method is used to prevent catastrophic forgetting in a NN with not enough capacity, in an experiment with 20 tasks. The evaluation accuracy of the first 5 tasks is shown.

methods), forgets gradually and linearly when the network capacity is too small to store that information.

This is shown in Figure 5.5. The experiment has been conducted on Permuted MNIST with 20 permutations on an MLP of two hidden layers of 20 neurons each. This is a very small network, and forgetting was expected.

The blue line, corresponding to the first task, forgets almost linearly during the training of the rest of the tasks. It also happens in some cases that when learning a new task, some accuracies drop catastrophically. This is a clear example of how regularization methods try to maintain previous knowledge. The *linear gradual forgetting* finishes over training step 50k, where the graph becomes more horizontal for all tasks. This suggests that at this point the network does not hold much task-specific information. The performance being better than random guess (0.1 accuracy) could be due to the shared representations of different tasks. However, as discussed in section 2.9.1, the permuted MNIST experiment is a very unrealistic benchmark where there are nearly no correlations between tasks. This leads to the belief that the only reason for accuracy to be above 10% is the task-specific multi-head classifier, which consists of a single fully connected layer per task.

The network in this experiment is extremely small, as it has been designed for forcing forgetting. The same experiment has been conducted with a wider network, consisting of 250 neurons per hidden layer instead of 20. All the other parameters are the same. The results in Figure 5.6 indicate that the network still forgets linearly, with a smaller slope than in the narrow network.

However, this linear forgetting, which might seem to be due to the regularization penalty, also appears when performing naive fine-tuning. Furthermore, with the appropriate hyperparameters, the results are even better than in Figure 5.6. *Would this happen in*

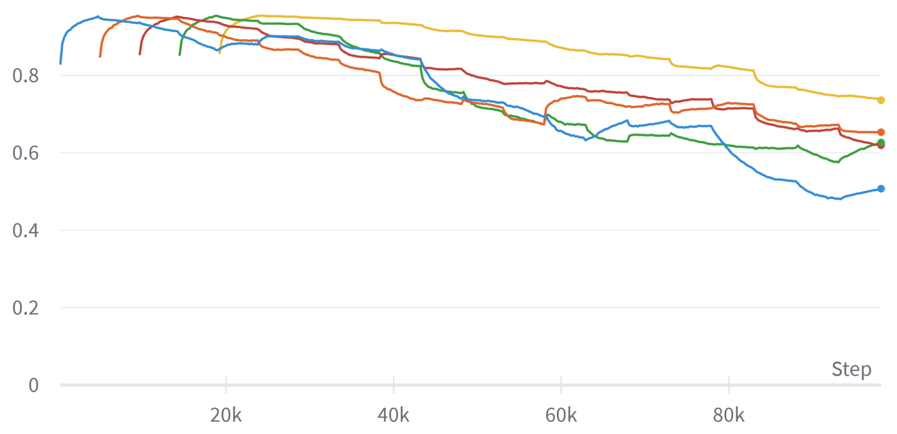


FIGURE 5.6: Same experiment of Figure 5.5 but with a wider network of width 250 per hidden layer. The evaluation accuracy of the first 5 tasks is shown.

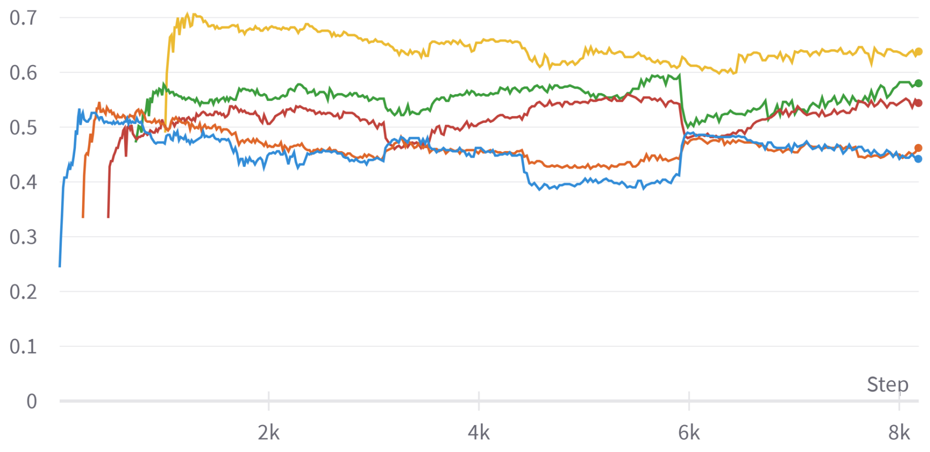


FIGURE 5.7: EWC method on split CIFAR-100 with the settings of subsection 5.2.3 does not forget.

more complex settings? is a question that naturally arises with these results. To answer this question, it has been observed how regularization acts on the accuracy of the first tasks with the same CL scenario as in the previous experiment with the CIFAR-100 dataset (subsection 5.2.3). Figure 5.7 shows the result. Although varying the multiplier m for the number of channels, forgetting does not occur with these settings.

The same experiment is made without regularization methods (Figure 5.8), and although the accuracies are slightly worse and there is a bit more of forgetting over time, the results are very similar to the ones obtained with regularization, and the average backward transfer is close to zero.

A conclusion of this experiment is that even if CL algorithms are effective to face catastrophic forgetting, it is often not enough, and proper architectures and hyperparameters must be taken for continual learning to be effective.

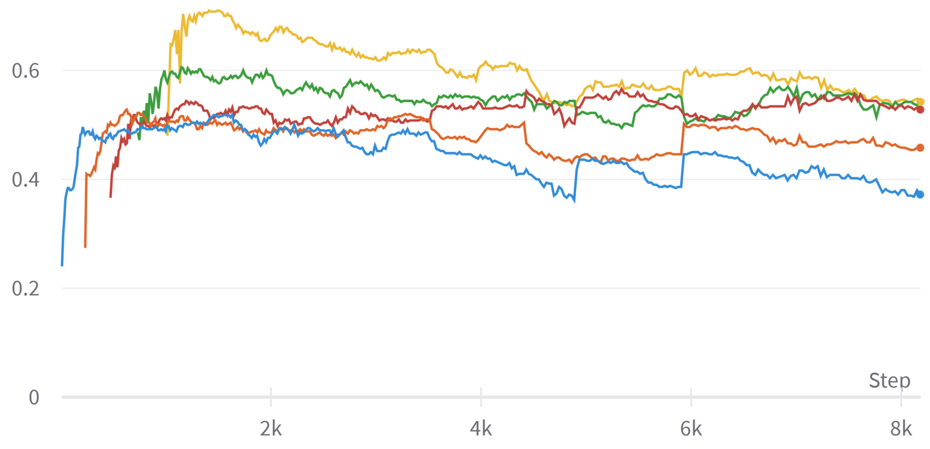


FIGURE 5.8: Naive fine-tuning on split CIFAR-100 with the settings from subsection 5.2.3.

5.4 Effectiveness of dropout as a regularization method

In a setting without memory-replay, *Stability-plasticity* dillema is usually faced with regularization methods. However, Seyed-Iman Mirzadeh, Farajtabar, and Hassan Ghasemzadeh, 2020 investigate this relationship and show that a stable network with dropout learns a gating mechanism such that for different tasks, different paths of the network are active.

In the following experiment, we ran naive fine-tuning on the MLP with two hidden layers of 100 units each, as in the other experiments with MLP. The architecture is single-head in order to have more forgetting. Therefore, the scenario is domain-incremental, as task-ids are not provided. The benchmark is Permuted MNIST, and the learning rate is fixed to 0.1. The results are in Figure 5.9 left and right. In the experiment, there are 10 tasks, but only the accuracy when evaluating on tasks 1 to 5 is shown. Although having dropout reduces forgetting at tasks 2, 3, and 4, the improvement is not clear. In fact, as has been shown in other experiments, in Continual Learning the correct selection of hyperparameters is game-changing, and it can make a method either have very good results or (in case of a bad selection) completely fail.

Following observations made by Seyed-Iman Mirzadeh, Farajtabar, and Hassan Ghasemzadeh, 2020, the hyperparameter that affects mainly accuracy and forgetting is the learning rate. Dropout is shown to be more effective with lower learning rate, but if the learning rate is too small, the *plasticity* of the network might not be enough. Therefore, applying dropout and reducing the learning rate at each task, and learning with 5 epochs per task, the following results were obtained:

As results show in Figure 5.10 left, the low plasticity problem is solved by having more training epochs per task. Stability is really good until learning experience 6, where all tasks start to experience catastrophic forgetting. However, performance is clearly better

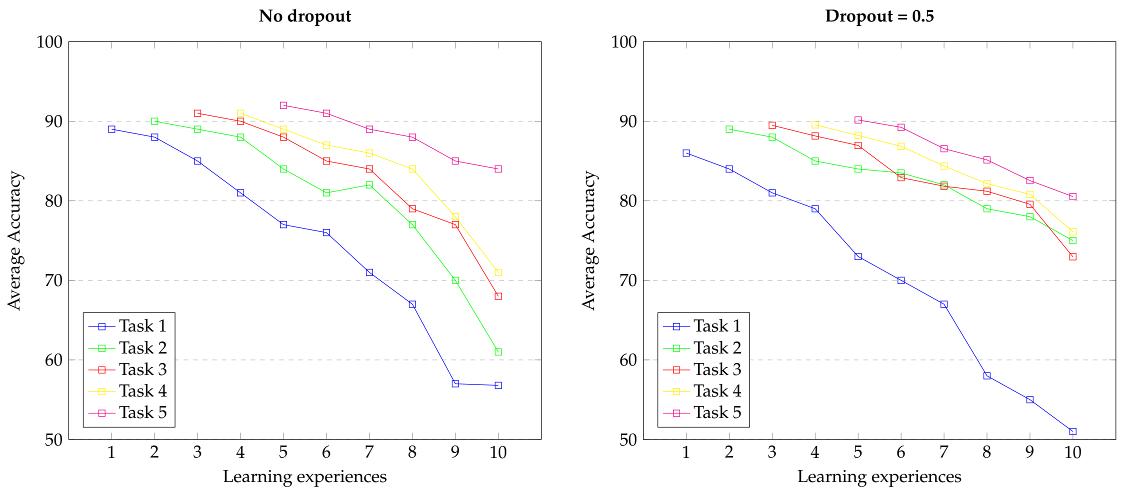


FIGURE 5.9

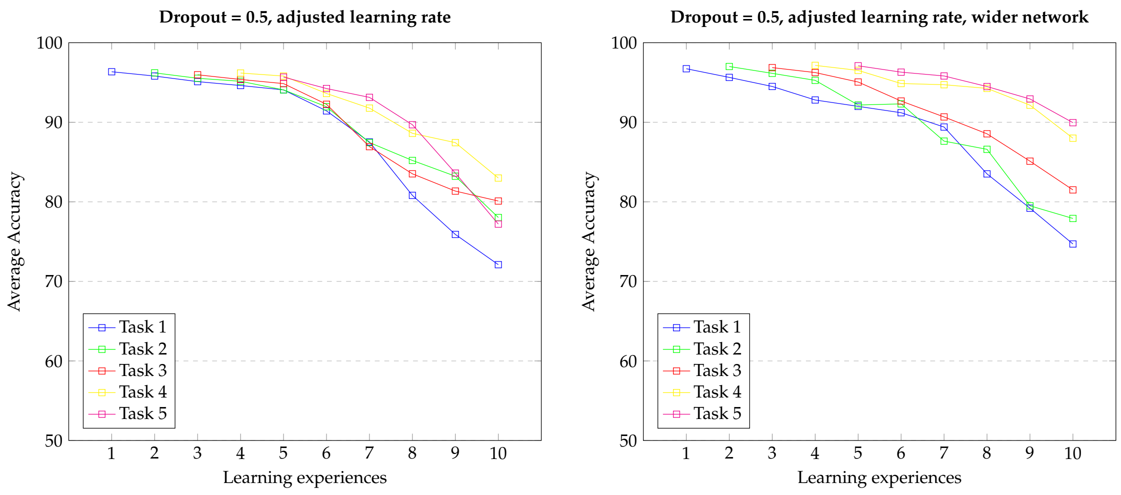


FIGURE 5.10

Dropout	BWT	Average acc.
No Dropout	-11.11	81.77
0.25	-9.60	84.69
0.4	-10.27	83.82
0.5	-8.03	**88.13**
0.6	**-6.99**	86.09
0.75	-10.05	84.28

TABLE 5.4: Average Backward Transfer of naive fine-tuning depending on the dropout probability

than without adjusting the learning rate. A reason for this loss from task 6 could be due to the network capacity. To test it, the same experiment with a wider network (with 400 units per hidden layer) has been carried out (Figure 5.10 right). The results improve slightly. Comparing first and last graphs of this section, we can conclude that (in simple scenarios) CL can be done just with naive learning, without an explicit algorithmic method to avoid catastrophic forgetting.

Additionally, Table 5.4 shows the Average Backward Transfer (explained at section 2.7) varying only the dropout, with the settings of Figure 5.10 right. This graph shows the dropout probability of 0.5, which has the best performance, as shown in the table.

Chapter 6

Conclusion & future directions

This thesis has mostly focused on understanding more deeply the nature of Continual Learning and providing insight on its phenomena. One of the main objectives is that someone who reads this thesis without any prior knowledge on CL can understand its context, current limitations, and most interesting research directions; thus being fully trained to contribute in the field. The experiments show that even if a big focus on CL is done on the algorithms, there is still a lot of research to do in the analysis of the architectures and optimizers, as they heavily affect Continual Learning. The results of the experiments published in the articles are often the best they have been able to achieve after investing a lot of effort in fine-tuning the experiment so that the outcomes are as favourable as possible. When other papers compare results with their own implementations or executions of the program, it is very common for there to be major disagreements about which methods are preferable. As an example, figure 6.1 shows the top-performance of A-GEM against other methods such as EWC. However, figure 6.2 shows that EWC has a better accuracy than A-GEM. Furthermore, A-GEM is shown to be the method with the worst performance in the table.

It is common to find in bibliography comparisons like these where results are very different from one another, even with simple benchmarks like permuted MNIST. This

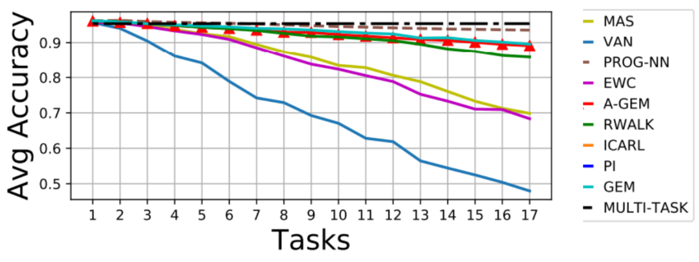
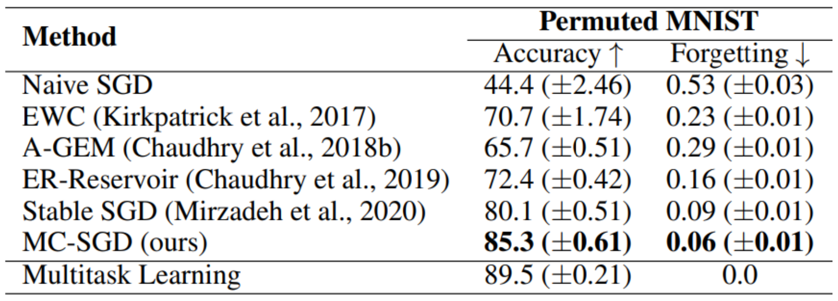


FIGURE 6.1: Figure from Chaudhry et al., 2018, which introduces Averaged GEM method. They compare multiple methods on permuted MNIST benchmark, and their method is top-performing, very close to the upper bound of multitask learning.



Method	**Permuted MNIST**	
	Accuracy ↑	Forgetting ↓
Naive SGD	44.4 (±2.46)	0.53 (±0.03)
EWC (Kirkpatrick et al., 2017)	70.7 (±1.74)	0.23 (±0.01)
A-GEM (Chaudhry et al., 2018b)	65.7 (±0.51)	0.29 (±0.01)
ER-Reservoir (Chaudhry et al., 2019)	72.4 (±0.42)	0.16 (±0.01)
Stable SGD (Mirzadeh et al., 2020)	80.1 (±0.51)	0.09 (±0.01)
MC-SGD (ours)	**85.3 (±0.61)**	**0.06 (±0.01)**
Multitask Learning	89.5 (±0.21)	0.0

FIGURE 6.2: Figure from Seyed-Iman Mirzadeh, Farajtabar, Görür, et al., 2020, which introduces Mode Connectivity - Stochastic Gradient Descent method. They compare in a table multiple methods on permuted MNIST benchmark, and A-GEM is the worst method, very far from multitask learning.

highlights the fact that further research is needed in order to find better ways of comparing results. A good library with tools and facilities to easily compare methods is needed, and improving the current framework of chapter 4 by adding new metrics and more recent methods is a good direction to continue. Currently, the library provides high flexibility to tune parameters and easily compare different models, but it still lacks of the implementation of some metrics explained at section 2.7. Additionally, the results on experiment at section 5.1 open up a new line of research on subspace connectivity.

Bibliography

Adam PyTorch (2019). URL: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html.

Aljundi, Rahaf et al. (Sept. 2018). "Memory Aware Synapses: Learning What (not) to Forget: 15th European Conference, Munich, Germany, September 8–14, 2018, Proceedings, Part III". In: pp. 144–161. ISBN: 978-3-030-01218-2. DOI: 10.1007/978-3-030-01219-9_9.

Allred, Jason M. and Kaushik Roy (2020). "Controlled Forgetting: Targeted Stimulation and Dopaminergic Plasticity Modulation for Unsupervised Lifelong Learning in Spiking Neural Networks". In: *Frontiers in Neuroscience* 14. ISSN: 1662-453X. DOI: 10.3389/fnins.2020.00007. URL: https://www.frontiersin.org/article/10.3389/fnins.2020.00007.

Carta, Antonio et al. (2021). "Ex-Model: Continual Learning from a Stream of Trained Models". In: *CoRR* abs/2112.06511. arXiv: 2112.06511. URL: https://arxiv.org/abs/2112.06511.

Chaudhry, Arslan et al. (2018). "Efficient Lifelong Learning with A-GEM". In: *CoRR* abs/1812.00420. arXiv: 1812.00420. URL: http://arxiv.org/abs/1812.00420.

Chizat, Lenaic, Edouard Oyallon, and Francis Bach (2018). "On Lazy Training in Differentiable Programming". In: DOI: 10.48550/ARXIV.1812.07956. URL: https://arxiv.org/abs/1812.07956.

Choi, Dami et al. (2019). "On Empirical Comparisons of Optimizers for Deep Learning". In: *CoRR* abs/1910.05446. arXiv: 1910.05446. URL: http://arxiv.org/abs/1910.05446.

ContinualAI, Course (2021). URL: https://course.continualai.org/lectures/scenarios-and-benchamarks.

ContinualAI, Wiki (2021). *What is continual / lifelong learning?* URL: https://wiki.continualai.org/the-continualai-wiki/introduction-to-continual-learning/#what-is-continual-lifelong-learning.

Cossu, Andrea, Marta Ziosi, and Vincenzo Lomonaco (2021). "Sustainable Artificial Intelligence through Continual Learning". In: *CoRR* abs/2111.09437. arXiv: 2111.09437. URL: https://arxiv.org/abs/2111.09437.

Deng, Jia et al. (2009). "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee, pp. 248–255.

Doan, Thang et al. (2022). *Efficient Continual Learning Ensembles in Neural Network Subspaces*. DOI: 10.48550/ARXIV.2202.09826. URL: https://arxiv.org/abs/2202.09826.

Duchi, John, Elad Hazan, and Yoram Singer (July 2011). "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12, pp. 2121–2159.

Farquhar, Sebastian and Yarin Gal (2018). *Towards Robust Evaluations of Continual Learning*. DOI: 10.48550/ARXIV.1805.09733. URL: https://arxiv.org/abs/1805.09733.

French, Robert M. (1999). "Catastrophic forgetting in connectionist networks". In: *Trends in Cognitive Sciences* 3.4, pp. 128–135. ISSN: 1364-6613. DOI: https://doi.org/10.1016/S1364-6613(99)01294-2. URL: https://www.sciencedirect.com/science/article/pii/S1364661399012942.

Goodfellow, Ian J. et al. (2015). *An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks*. arXiv: 1312.6211 [stat.ML].

Hayes, Tyler L. et al. (2018). "New Metrics and Experimental Paradigms for Continual Learning". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 2112–21123. DOI: 10.1109/CVPRW.2018.00273.

He, Kaiming et al. (2015). "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

Hemati, Hamed, Marco Schreyer, and Damian Borth (2021). "Continual Learning for Unsupervised Anomaly Detection in Continuous Auditing of Financial Accounting Data". In: *CoRR* abs/2112.13215. arXiv: 2112.13215. URL: https://arxiv.org/abs/2112.13215.

Hinton, Geoff (2012). *Lecture 6a. Overview of mini-batch gradient descent*. URL: https://www.cs.toronto.edu/~tijmen/csc321//slides/lecture_slides_lec6.pdf.

Hong, DaeYong, Yan Li, and Byeong-Seok Shin (2019). "Predictive EWC: mitigating catastrophic forgetting of neural network through pre-prediction of learning data". In: *Journal of Ambient Intelligence and Humanized Computing*. ISSN: 1868-5145. DOI: 10.1007/s12652-019-01346-7. URL: https://doi.org/10.1007/s12652-019-01346-7.

Hsu, Yen-Chang, Yen-Cheng Liu, and Zsolt Kira (2018). "Re-evaluating Continual Learning Scenarios: A Categorization and Case for Strong Baselines". In: *CoRR* abs/1810.12488. arXiv: 1810.12488. URL: http://arxiv.org/abs/1810.12488.

IBM, Cloud Education (2020). *What is unsupervised learning?* URL: https://www.ibm.com/cloud/learn/unsupervised-learning.

Jacot, Arthur, Franck Gabriel, and Clément Hongler (2018). "Neural Tangent Kernel: Convergence and Generalization in Neural Networks". In: *CoRR* abs/1806.07572. arXiv: 1806.07572. URL: http://arxiv.org/abs/1806.07572.

Kaplanis, Christos, Murray Shanahan, and Claudia Clopath (2018). "Continual Reinforcement Learning with Complex Synapses". In: *CoRR* abs/1802.07239. arXiv: 1802.07239. URL: http://arxiv.org/abs/1802.07239.

Ke, Zixuan, Bing Liu, and Xingchang Huang (2021). “Continual Learning of a Mixed Sequence of Similar and Dissimilar Tasks”. In: *CoRR* abs/2112.10017. arXiv: 2112.10017. URL: https://arxiv.org/abs/2112.10017.

Kingma, Diederik P. and Jimmy Ba (2014). *Adam: A Method for Stochastic Optimization*. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. URL: http://arxiv.org/abs/1412.6980.

Kirkpatrick, James et al. (2016). “Overcoming catastrophic forgetting in neural networks”. In: *CoRR* abs/1612.00796. arXiv: 1612.00796. URL: http://arxiv.org/abs/1612.00796.

Kornblith, Simon et al. (2019). “Similarity of Neural Network Representations Revisited”. In: *CoRR* abs/1905.00414. arXiv: 1905.00414. URL: http://arxiv.org/abs/1905.00414.

Krizhevsky, Alex (2009). *Learning multiple layers of features from tiny images*. Tech. rep.

Lecun, Yann (1998). *The mnist database*. URL: http://yann.lecun.com/exdb/mnist/.

Lesort, Timothée et al. (2019). “Continual Learning for Robotics”. In: *CoRR* abs/1907.00182. arXiv: 1907.00182. URL: http://arxiv.org/abs/1907.00182.

Li, Da et al. (2016). “Evaluating the Energy Efficiency of Deep Convolutional Neural Networks on CPUs and GPUs”. In: *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, pp. 477–484. DOI: 10.1109/BDCloud-SocialCom-SustainCom.2016.76.

Liu, Sue (Jan. 2019). *Dirichlet distribution*. URL: https://towardsdatascience.com/dirichlet-distribution-a82ab942a879.

Lomonaco, Vincenzo and Davide Maltoni (2017). “CORe50: a New Dataset and Benchmark for Continuous Object Recognition”. In: *CoRR* abs/1705.03550. arXiv: 1705.03550. URL: http://arxiv.org/abs/1705.03550.

Lomonaco, Vincenzo, Lorenzo Pellegrini, et al. (2021). “Avalanche: an End-to-End Library for Continual Learning”. In: *CoRR* abs/2104.00405. arXiv: 2104.00405. URL: https://arxiv.org/abs/2104.00405.

Lopez-Paz, David and Marc’Aurelio Ranzato (2017). “Gradient Episodic Memory for Continuum Learning”. In: *CoRR* abs/1706.08840. arXiv: 1706.08840. URL: http://arxiv.org/abs/1706.08840.

Maass, Wolfgang (1997). “Networks of spiking neurons: The third generation of neural network models”. In: *Neural Networks* 10.9, pp. 1659–1671. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(97)00011-7. URL: https://www.sciencedirect.com/science/article/pii/S0893608097000117.

Mai, Zheda et al. (2021). “Online Continual Learning in Image Classification: An Empirical Survey”. In: *CoRR* abs/2101.10423. arXiv: 2101.10423. URL: https://arxiv.org/abs/2101.10423.

Mallya, Arun and Svetlana Lazebnik (2017). "PackNet: Adding Multiple Tasks to a Single Network by Iterative Pruning". In: *CoRR* abs/1711.05769. arXiv: 1711.05769. URL: http://arxiv.org/abs/1711.05769.

Maltoni, Davide and Vincenzo Lomonaco (Apr. 2019). "Continuous learning in single-incremental-task scenarios". In: *Neural Networks* 116. DOI: 10.1016/j.neunet.2019.03.010.

McCloskey, Michael and Neal J. Cohen (1989). "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem". In: ed. by Gordon H. Bower. Vol. 24. Psychology of Learning and Motivation. Academic Press, pp. 109–165. DOI: https://doi.org/10.1016/S0079-7421(08)60536-8. URL: https://www.sciencedirect.com/science/article/pii/S0079742108605368.

Meyer, Bertrand (1997). *Object-Oriented Software Construction*. 2nd ed. Upper Saddle River, NJ: Prentice Hall. ISBN: 978-0-13-629155-8.

Mirzadeh, S. and H. Ghasemzadeh (June 2021). "CL-Gym: Full-Featured PyTorch Library for Continual Learning". In: *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 3616–3622. DOI: 10.1109/CVPRW53098.2021.00401. URL: https://doi.ieeecomputersociety.org/10.1109/CVPRW53098.2021.00401.

Mirzadeh, Seyed Iman et al. (2021). "Wide Neural Networks Forget Less Catastrophically". In: *ArXiv* abs/2110.11526.

Mirzadeh, Seyed-Iman, Arslan Chaudhry, et al. (2022). "Architecture Matters in Continual Learning". In: *CoRR* abs/2202.00275. arXiv: 2202.00275. URL: https://arxiv.org/abs/2202.00275.

Mirzadeh, Seyed-Iman, Mehrdad Farajtabar, and Hassan Ghasemzadeh (2020). "Dropout as an Implicit Gating Mechanism For Continual Learning". In: *CoRR* abs/2004.11545. arXiv: 2004.11545. URL: https://arxiv.org/abs/2004.11545.

Mirzadeh, Seyed-Iman, Mehrdad Farajtabar, Dilan Görür, et al. (2020). "Linear Mode Connectivity in Multitask and Continual Learning". In: *CoRR* abs/2010.04495. arXiv: 2010.04495. URL: https://arxiv.org/abs/2010.04495.

Mnih, Volodymyr et al. (2013). "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602. arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

Morley, Jessica et al. (2020). "From What to How: An Initial Review of Publicly Available AI Ethics Tools, Methods and Research to Translate Principles into Practices". In: *Science and Engineering Ethics* 26.4, pp. 2141–2168. ISSN: 1471-5546. DOI: 10.1007/s11948-019-00165-5. URL: https://doi.org/10.1007/s11948-019-00165-5.

Normandin, Fabrice et al. (2021). "Sequoia: A Software Framework to Unify Continual Learning Research". In: *CoRR* abs/2108.01005. arXiv: 2108.01005. URL: https://arxiv.org/abs/2108.01005.

Parisi, German Ignacio et al. (2018). "Continual Lifelong Learning with Neural Networks: A Review". In: *CoRR* abs/1802.07569. arXiv: 1802.07569. URL: http://arxiv.org/abs/1802.07569.

Rao, Dushyant et al. (2019). “Continual Unsupervised Representation Learning”. In: *CoRR* abs/1910.14481. arXiv: 1910.14481. URL: http://arxiv.org/abs/1910.14481.

Rebuffi, Sylvestre-Alvise, Alexander Kolesnikov, and Christoph H. Lampert (2016). “iCaRL: Incremental Classifier and Representation Learning”. In: *CoRR* abs/1611.07725. arXiv: 1611.07725. URL: http://arxiv.org/abs/1611.07725.

Riemer, Matthew et al. (2018). “Learning to Learn without Forgetting By Maximizing Transfer and Minimizing Interference”. In: *CoRR* abs/1810.11910. arXiv: 1810.11910. URL: http://arxiv.org/abs/1810.11910.

Ritter, Hippolyt, Aleksandar Botev, and David Barber (2018). *Online Structured Laplace Approximations For Overcoming Catastrophic Forgetting*. DOI: 10.48550/ARXIV.1805.07810. URL: https://arxiv.org/abs/1805.07810.

Ruder, Sebastian (2016). “An overview of gradient descent optimization algorithms”. In: *CoRR* abs/1609.04747. arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

Russell, Stuart J. (Stuart Jonathan) (2010a). *Artificial intelligence : a modern approach*. Includes bibliographical references (pages 1063-1093) and index. Third edition. Upper Saddle River, N.J. : Prentice Hall, [2010] ©2010. URL: https://search.library.wisc.edu/catalog/9910082172502121.

– (2010b). *Artificial intelligence : a modern approach*. Includes bibliographical references (pages 1063-1093) and index. Third edition. Upper Saddle River, N.J. : Prentice Hall, [2010] ©2010. URL: https://search.library.wisc.edu/catalog/9910082172502121.

Rusu, Andrei A. et al. (2016). “Progressive Neural Networks”. In: *CoRR* abs/1606.04671. arXiv: 1606.04671. URL: http://arxiv.org/abs/1606.04671.

Serrà, Joan et al. (2018). “Overcoming catastrophic forgetting with hard attention to the task”. In: *CoRR* abs/1801.01423. arXiv: 1801.01423. URL: http://arxiv.org/abs/1801.01423.

SGD PyTorch (2019). URL: https://pytorch.org/docs/stable/generated/torch.optim.SGD.html.

Welling, Max (Jan. 2009). “Herding dynamical weights to learn”. In: vol. 382, p. 141. DOI: 10.1145/1553374.1553517.

Wortsman, Mitchell et al. (2021). “Learning Neural Network Subspaces”. In: *CoRR* abs/2102.10472. arXiv: 2102.10472. URL: https://arxiv.org/abs/2102.10472.

Wynsberghe, Aimee (Feb. 2021). “Sustainable AI: AI for sustainability and the sustainability of AI”. In: *AI and Ethics* 1. DOI: 10.1007/s43681-021-00043-6.

Zenke, Friedemann, Ben Poole, and Surya Ganguli (2017). “Improved multitask learning through synaptic intelligence”. In: *CoRR* abs/1703.04200. arXiv: 1703.04200. URL: http://arxiv.org/abs/1703.04200.

Zhou, Pan et al. (2020). “Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning”. In: *CoRR* abs/2010.05627. arXiv: 2010.05627. URL: https://arxiv.org/abs/2010.05627.

Zhuang, Fuzhen et al. (2019). “A Comprehensive Survey on Transfer Learning”. In: *CoRR* abs/1911.02685. arXiv: 1911.02685. URL: http://arxiv.org/abs/1911.02685.

Appendix A

Further background

A.1 Cross-entropy loss

Cross-entropy is a measure from the field of Information Theory. It is a measure of the difference between two probability distributions.

Cross-entropy loss, also called log loss, between the true probability function P^* and the predicted probability function P conditioned to the input sample $x_i \in \mathcal{X}$ is

$$H(P^*|P) = -\sum_{y \in \mathcal{Y}} P^*(y|x_i) \log P(y|x_i, \theta)$$

where θ corresponds to the network's parameters. Therefore if the correct output for x_i is y_i,

$$P^*(y) = \begin{cases} 0 & y \neq y_i \\ 1 & y = y_i \end{cases}$$

and thus $H(P^*|P) = -\log P(y_i|x_i, \theta)$. Note that in the case where the agent has information about the task-id, $H(P^*|P) = -\log P(y_i|x_i, t_i, \theta)$. Minimizing the cross-entropy loss is equivalent to minimizing the Kullback–Leibler divergence $D_{KL}(P^*||P)$ between the two distributions. For more insight on KL divergence, visit https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

A.2 I.i.d. hypothesis

In probability, a set of random variables, the input samples in our case, are independent and identically distributed if they are sampled from the same probability distribution, and all are mutually independent.

Identically Distributed means that there are no overall trends. The distribution does not fluctuate and all items are taken from the same probability distribution. This does not happen in Continual Learning, as each task has its own distribution.

Independent means that the items are all independent events. In other words, they are not connected to each other in any way. Knowledge of the value of one variable (one input image) gives no information about the value of the other and vice versa.