



UNIVERSITÀ
DEGLI STUDI
FIRENZE

PHD PROGRAM IN SMART COMPUTING
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE (DINFO)

Learning from Video Streams: Virtual Environments and Parallel Computation

Enrico Meloni

Dissertation presented in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Smart Computing

*PhD Program in Smart Computing
University of Florence, University of Pisa, University of Siena*

Learning from Video Streams: Virtual Environments and Parallel Computation

Enrico Meloni

Advisor:

Prof. Marco Gori

Head of the PhD Program:

Prof. Stefano Berretti

Evaluation Committee:

Prof. Roberto Giorgi, *University of Siena*

Prof. Oswald Lanz, *Free University of Bozen-Bolzano*

Prof. Marco La Cascia, *University of Palermo*

Acknowledgments

At the end of this three-year-long journey, I would like to say thanks to Marco Gori for accepting to be my supervisor, welcoming me in SAILab and for guiding me all this time and for providing the many ideas that helped shaping my research during the PhD. I would also like to thank the professors that have been a part of my Supervisory Committee: Prof. Carlo Sansone, Prof. Vincent Lepetit, and Stefano Merler. I would also like to thank Prof. Battista Biggio and Prof. Bing Liu for taking the time to review my thesis and for the useful suggestions. I would also like to thank Stefano Melacci for the dedication in following the projects on which I worked, giving me valuable insights.

I extend my thanks to the administrative staff of both Siena and Florence universities, who greatly helped me through the bureaucracy needed for my activities, with a particular mention to Simona Altamura and her quick and punctual responses. I thank the BrainControl branch of LiquidWeb S.r.l. for hosting me during my internship. I also thank the Regione Toscana for the Pegaso grant that funded my PhD.

My sincere thanks go to the members of SAILab, with whom I shared these three years. Even if during half of this period we have been apart due to COVID, I think we managed to form a close bond that I hope will remain even after the end of my PhD. Thanks to Matteo, for appreciating the fine art of memes, for the interesting discussions during coding sessions, and for reassuring me when I felt anxious for the many difficulties. Thanks to Andrea, for his many suggestions from even before starting the PhD, for the stimulating talks while having lunch at Gino Cacino, but mainly for passing me the honor of administrating the SAILab Website. Thanks to Francesco, for being so welcoming from day one, for the many nerdy nights with Magic and Heroquest, and for all your significant help in these years. Thanks to Lapo for the fun conversations, for sharing the same difficulties and taking them on together, and for teaching me how to snowboard. Thanks to Niccolò, for having welcomed me in Siena, for the coffee breaks, and for all the fun times together. Thanks to Alessandro, for his help when I could not understand what was breaking my \LaTeX codes, and for the engaging conversations on Discord about cooking, chocolate, and Jurassic Park. Thanks to Stefano F., for making me laugh about my moments of stress and for inviting me at soccer games. Thanks to the other members of Lab 201, Simone M., Michele, Gabriele, Lisa, Dario, Luca, and Giuseppe for sharing this experience with me and for the nice dinners and activities together. Thanks also to the members of Lab 202, Simone B., Paolo, Pietro, Alberto, Giorgia, Giorgio, Anna, Alessio, Caterina, Veronica, Filippo, Elia, Barbara, Giovanna, and Federica for letting me randomly barge into the lab to have a chat with all of you, and to some of you for the fun games of tennis. I also want to thank all professors involved in

SAILab, Monica Bianchini, Franco Scarselli and Marco Maggini, for welcoming me and helping me in various occasions.

I want to thank my friends outside of the PhD for many things: for being supportive, for giving me the occasions of relaxing and chilling, for sharing many experiences in our life, for the fun times, activities and chats. Thanks to Tiziana, Paola, and Daniele for having been in my life for all these years, we have a special bond that I really treasure and wish to maintain for even longer. Thanks to Raffaele for having accompanied me during the University years, in our projects, our coding challenges with Marco, and with our talks about our professional future. Thanks to the friends I met in Pisa: Mirko, Maurizio, Dario, Eugenia, I miss the times we spent together. Thanks to all my friends from Sardegna, with whom I am so happy to have kept in contact during all these years I have been away: Eli, Gigi, Gianluca, Ilaria, Andrea, Gaia, Silvia, Oliver, Laura, Silvio. Thanks to Sofia, Michele, Andrea, Maria, Martina, and Gioele for the fun nights at the pub. Thanks to the friends that kept me company and helped me get through the lockdown by playing online together: Marta, Tina, Mas, Elena, and Chiara. Thanks to Assunta and Martina for creating a wonderful space thanks to which I met many beautiful people during the last year: Bb, Tommaso, Fausto, and many others. I want to thank Bb for the wonderful time together, for the trust they have for me, for the mutual support and help, and for the unvaluable reassurance they gave me during these difficult times.

Of course, none of this would have been possible without the unreplaceable help and support of my family, always present and attentive to my needs and supportive for my future. In particular, thanks mom, dad and Luca.

Abstract

Researchers have always been fascinated by the idea of developing computer programs that could replicate innate human abilities such as language or vision. Recently, the Machine Learning community has increased its efforts towards Continual and Lifelong Learning, pursuing the ambitious objective of developing autonomous learning agents that learn similarly to humans. Research in this direction highlights the strikingly artificial approach that has been followed until now in Machine Learning, where the learning procedures dictates the use of huge datasets and that the learner is shown shuffled samples with no particular correlation among them and sampled from a static dataset. Clearly, this is significantly different from what humans and animals experience in the real world, that is a continuous multi-sensory stream extracted from a dynamical environment, with correlations among each modality of the stream, but also between consecutive samples in each stream, where the flow of time has a central importance in the way the environment is experienced by the learning agents. The need of suitable environments in which an artificial learning agent can live and learn has driven the Artificial Intelligence community to design and implement 3D physical simulations, called Virtual Environments, that straightforwardly offer a dynamical environment with the capability of creating agents that interface with learning algorithms and can experience their surroundings and interact with it. However, currently available solutions are not mature enough to fully implement Lifelong Learning agents, with environments that remain fundamentally static with the exception of interactions by the learning agent. Furthermore, up until now there has been little research on the safety and security of such Virtual Environments with respect to malicious users that wish to poison or undermine the integrity of Virtual Environments to damage the learning of agents living inside them. Finally, while there has been abundant research on accelerating traditional batch-mode offline learning, little research has been produced on the matter of accelerating real-time online learning, which is needed by a learning agent perceiving a real-time online sensory stream. In this thesis, we address these open problems on three fronts, i. e. real-time stream generation, safety and security to Adversarial Attacks, acceleration of real-time online learning. We introduce SAILenv, a platform specifically designed to allow real-time generation and perception of visual streams, with powerful features of parametrical generation of scenarios aimed at creating incrementally complex streams of data, specifically considering Continual Learning tasks; we study the safety and security of the graphical generation engines of the available Virtual Environments, showing that it is possible to implant Adversarial 3D Objects able to poison all scenarios in which such objects are integrated; finally, we introduce PARTIME, a library specifically designed for online real-time learners, that must complete processing of a sample from a stream before the next sample is made available, to maintain real-time performances.

Contents

Contents	1
1 Introduction	3
1.1 Motivation	3
1.2 Contributions	8
1.3 Structure of the Thesis	11
2 Background	13
2.1 Virtual Environments	13
2.2 Continual and Lifelong Learning	20
2.3 Renderers	23
2.4 Adversarial Attacks	26
2.5 Parallelism	30
3 Making Virtual Environments Simple: SAILenv	35
3.1 Related Platforms	38
3.2 Architecture	40
3.3 Details on the generated views	42
3.4 Photo-realistic Objects and Scenes	45
3.5 Dynamical Objects and Moving Agent	47
3.6 Using SAILenv	48
3.7 Experimental Evaluation	50
3.8 Discussion	53
4 Dynamic Virtual Environments for Continual Machine Learning	57
4.1 Parametric Generation of Environments	60
4.2 Continual Learning 3D Virtual Benchmark	62
4.3 Examples	66
4.4 Discussion	69
5 Adversarial Attacks in Virtual Environments	71
5.1 Renderers: Differentiable and Non-Differentiable	73

5.2	Adversarial Attacks and Adversarial 3D Objects	74
5.3	Experimental study	79
5.4	Discussion	86
6	Parallel Computations in Learning from a Video Stream	89
6.1	Related Work	91
6.2	Scalable and Parallel Local Computations Over Time	95
6.3	PARTIME	100
6.4	Experiments	103
6.5	Discussion	110
7	Conclusions	115
7.1	Summary of Contributions	115
7.2	Future work	117
A	Publications	119
	Bibliography	123

Chapter 1

Introduction

1.1 Motivation

Researchers have always been fascinated by the idea of developing computer programs that could replicate innate human abilities such as language or vision (Thrun and Mitchell, 1995; Hassabis et al., 2017). This aim has been the driving force for many research works since the late 1950, when the computational capabilities of hardware started to increase and allow researchers to develop and test new Machine Learning algorithms. A good tractation of the early years of machine learning can be found in (Fradkov, 2020). Here we summarize some of the most interesting points in the history for the scope of the thesis. It was in (Rosenblatt, 1958) that the Perceptron was first formalized, introducing the Neural Network architectures. Unfortunately, after a little more than a decade, it became clear that the computational power available was still not enough to tackle real-world problems with these newly found algorithms, and thus in the early 1970s a period of reduced interest started, and it was called the first *AI Winter*.

Later, in the early 1980s, the pursuit for better Artificial Intelligence was resumed with new developments in algorithms and toolkits, such as the introduction of Backpropagation in (Rumelhart et al., 1985, 1986). Together with computational advances, the successes of Backpropagation brought forth many new research work on the topic of Machine Learning. Unfortunately, successes on real-world problems were less than what was hoped; furthermore, some drawbacks were found in the use of Backpropagation, see (Brady et al., 1989; Gori and Tesi, 1992). This led to the second *AI Winter* in the early 1990s.

After a decade of apparent silence and low interest, the first decade of the 21st century, laid the stepping stones which would enable the current *AI Spring* and the successes on real-world problem that we can see all around us. In fact, we can identify three parallel trends that helped build the foundations of current AI research: *Data Availability*, *Deep Neural Models*, and *Parallel Computing*.

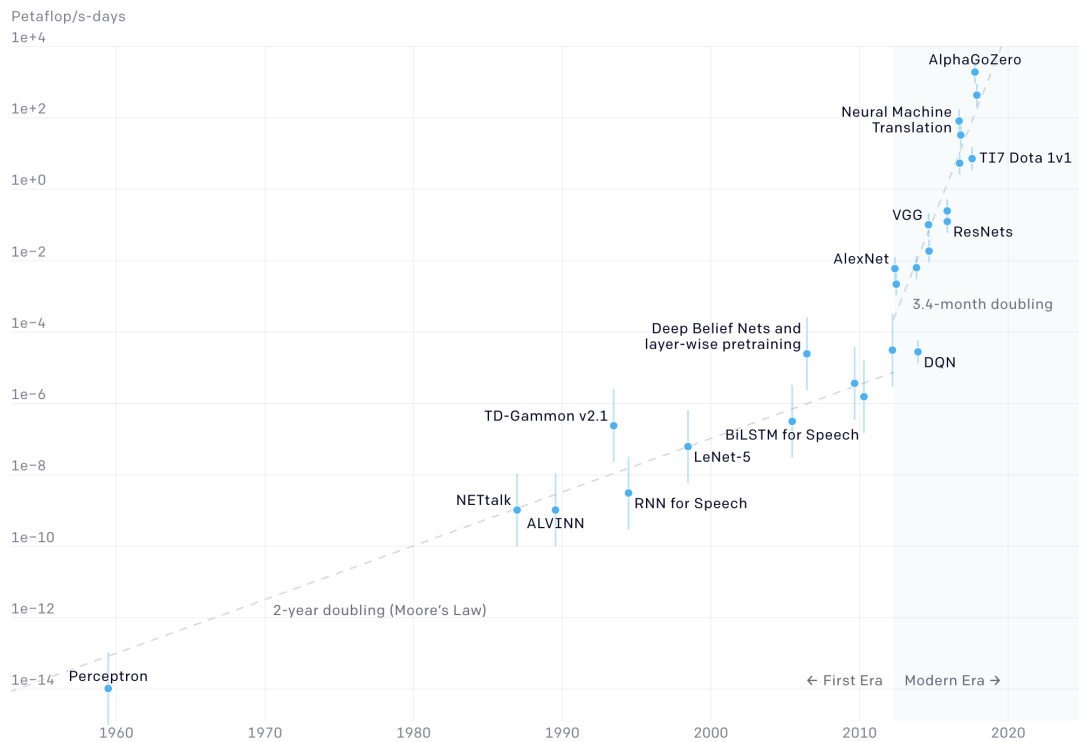


Figure 1.1: Growth of neural models computational complexity, expressed in petaflop/s-day, through the last 60 years. Petaflop/s-day amounts to the number of operations accumulated by performing a petaflop per second for one day, equivalent to 10^{20} flops. Here we define flop as FLOat OPeration. Float operations per seconds are instead expressed as flop/s instead of FLOPS as it is expressed in other texts. Image courtesy of OpenAI (<https://openai.com/blog/ai-and-compute/>).

Data Availability. With the advent of the internet, social networks, web archives, and others, the research community had access to huge quantities of data, that would prove useful in the training and testing of new algorithms, allowing the creation of standardized benchmarks on which to demonstrate new advances and contributions. An exemplary case of such trend is ImageNet (Deng et al., 2009), a large scale image dataset, annotated with crowd-sourcing to tackle the great cost of manual annotations of such a huge number of images. Every year since 2010, the ImageNet Large Scale Visual Recognition Challenge (Russakovsky et al., 2015) has gathered new contributions and encouraged new breakthrough research. Considering even other fields of Machine Learning, data in this era is more accessible and abundant as ever. The Web is also a great source of textual data (Liu and Curran, 2006), and indeed it was used in many works in the field of Natural Language Pro-

cessing. Recent works in the field (Radford et al., 2019; Brown et al., 2020) leverage the availability of enormous textual datasets to perform unsupervised pre-training to reduce labeling costs and therefore allow the training of huge models.

Deep Neural Models. The basic idea of the Perceptron was brought back and developed, introducing the concept of Deep Machine Learning. Out of the many applications of Deep Learning, we mention Convolutional Neural Networks or CNN, introduced in (LeCun et al., 1989a,b), since a CNN called AlexNet (Krizhevsky et al., 2017) won the ImageNet challenge in 2012 with a huge margin to its next competitor, demonstrating the learning capabilities expressed by these new algorithms and bringing further research interest in the field of Deep Learning and Convolutional Neural Networks. Since then, researchers have increased the complexity of their neural models at an even faster pace, as summarized in Figure 1.1. In (Simonyan and Zisserman, 2014), VGG networks were presented, deep networks with 16-19 weight layers, and again winning the ImageNet competition. In (He et al., 2016), Residual Networks were introduced, increasing the depth of the network by 8 times with respect to VGG, showing how deep networks can increase the benefits and better optimize learning by winning the 2015 ImageNet competition. Recently, a class of neural models known as *Transformers*, introduced in (Vaswani et al., 2017), has been a driving force in increasing model complexity to unprecedented scales. This is the example of GPT-2 (Radford et al., 2019), with around 1.5 billion parameters and GPT-3 (Brown et al., 2020), with around 175 billion parameters. In general, the trend of recent research has been that of increasing model complexity to better take advantage of the huge set of data available on the internet.

Parallel computing and memory. During the first decade of the 21st century, many breakthroughs were unveiled in the field of parallel computing, for example Google MapReduce in 2004 and Hadoop in 2006, enabling new software that distributed processing among multiple processors while efficiently exchanging huge amounts of data between processors. At the same time, the GPUs market, which was originally intended for 3D graphics and videogames, had a new breakthrough by NVIDIA, which developed general purpose GPUs and the CUDA language that allowed to write software in a C-like language. This of course allowed researchers to implement learning algorithms in a parallel fashion through the use of GPUs innate parallel capabilities (Steinkraus et al., 2005; Chellapilla et al., 2006; Cireşan et al., 2010). The conjunction of GPUs and Machine Learning allowed both fields to develop at a quick pace, bringing forth novel tools that enabled further research on the topics, such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019). The availability of tools that accelerate the computation for neural networks allowed the research community to increase the complexity of models. Unfortunately, the limit

of single devices in terms of achievable computational speed and available memory was quickly reached, forcing researchers to intersect with the field of Distributed Computing, designing networks and algorithms that can operate on networks of devices, taking advantage of concurrently executing parallelizable operations (e. g. most of the training loop is parallelizable over independent data), more available memory and fast data exchange protocols (Shoeybi et al., 2019).

The current landscape of Machine Learning research has deeply evolved during the recent decades, leveraging the availability of huge datasets, the increasing complexity of neural models alongside the hardware and technical advances for enabling such huge models. However, many evaluation protocols, such as ImageNet (Russakovsky et al., 2015), were designed as a static evaluation: there is a fixed training dataset and a fixed testing dataset, the two are accurately separated as the second is only used for the final evaluation. The training dataset is often composed of independent samples of the considered categories, and they are almost always presented to the network in shuffled batches (Lomonaco and Maltoni, 2017). In general, the main focus of past research was towards batch-mode offline learning on huge datasets. Recently, however, the Machine Learning community has increased the efforts towards researching *Continual* and *Lifelong* Learning, due to the interest towards autonomous learning agents. These recent works highlight the strikingly artificial approach that is currently being used for the development of learning models. In fact, the shuffled independent samples in datasets present significantly different learning conditions with respect to those to which humans and other animals are exposed during their lifetime (Krueger and Dayan, 2009; Cangelosi and Schlesinger, 2018). What humans experience is a continuous multimodal stream of data, experienced and processed in real-time, online, without storing huge collections of data and re-experiencing them in random order. In this thesis, by *real-time stream* we intend a stream of data S where the interarrival times between a sample and the next one are always under a certain threshold Δ_S . Often, when considering online learning from a real-time stream, it is considered of most importance to keep the output stream O framerate at least equal to the input stream I framerate. This can be framed as $\text{fps}_O \geq \text{fps}_I$, where fps_O and fps_I are the output and the input stream framerates respectively. This is equivalent to $\frac{1}{\Delta_O} \geq \frac{1}{\Delta_I}$ which finally sets the threshold $\Delta_O \leq \Delta_I$. When considering sequential processing, Δ_O is almost always equal to the response time of a network. However, taking into account parallel processing, Δ_O can be actually lower than the response time of the network, which can help speeding up computations in settings where the processing delay can be considered unimportant. This will be considered in more depth in chapter 6.

Recent works, such as (Lomonaco and Maltoni, 2017) made step forwards into the definition of datasets and benchmarks which accept “*episodes*”, that is short videos where temporal coherence is maintained and the agent is supposed to learn

from a continuous source of data, slowly integrating new information from new episodes. While this is certainly an improvement with respect to the common offline batch-mode approach, we argue that this is still not close enough to what humans perceive and elaborate. To fully achieve the idea of an embodied agent that learns as long as it lives, we need a suitable environment in which to let it discover and extract new information from what it can sense. Unfortunately, for many applications, such as robotics, using the real-world as the chosen environment is unfeasible for several reasons, for instance the cost and the dangers to which humans or things could be exposed. To solve some of these issues, 3D Physical simulations, also called *Virtual Environments*, were introduced by the Machine Learning community, to allow the generation of experimental settings, often with full automated labeling, at low cost and with negligible risk. In (Shu et al., 2021) an episodic benchmark called AGENT is presented, leveraging the physical simulation of ThreeDWorld (Gan et al., 2020) to automatically generate a great amount of visual data, and in (Shridhar et al., 2020), the authors introduce a benchmark for visual navigation and instructions interpretation, called ALFRED, generated through the AI2-THOR environment (Kolve et al., 2017). We argue that Virtual Environments are actually a fundamental tool to fulfill the ambition of Lifelong Learning agents that can live inside these physical simulations, experiencing multimodal sensory information, under the physical constraints of the environment, in a virtually endless timeline with the potential of slowly introducing new difficulties and concepts to the agent. Some work in this direction has been started in RoboTHOR (Deitke et al., 2020), which use the AI2THOR Virtual Environment (Kolve et al., 2017) to generate a real-time visual stream from an indoor environment with interactable entities. However, the environment itself is still fundamentally static in nature, as the number of potential interactions is still limited by the learning agent capabilities, most of the objects are not in movement and nothing will change without the intervention of the learning agent.

While Virtual Environments are indeed a powerful tool that is most likely suitable to enable further research in the field of Continual and Lifelong Learning, there is currently little research on their reliability and robustness to attacks that aim at invalidating the results of benchmarks through them. It is known that Adversarial Attacks are also performable on 3D objects (Athalye et al., 2018), but not enough studies have focused on the transferability of such attacks to Virtual Environments. This is an important issue that cannot be overlooked, especially when considering the case of public Virtual Environments that accept contributions from a wide community of people. In fact, once an Adversarial 3D Object is integrated into the environment, it can potentially poison many scenarios and each of the frames in the generated visual streams, exponentially spreading its malicious effect. Such adversarial objects could be used to lower the performances of a given model by crafting 3D objects aimed specifically at that model. They could also be used to inject back-

doors that can help a properly tuned model, similarly to what is already done for 2D images (Chen et al., 2018).

Regarding the computational side, Continual and Lifelong Learning in the online real-time processing paradigm, are limited by the achievable computational speed of current hardware. As a matter of fact, current neural models are so deep that real-time processing, intended as processing each sample of a stream before the arrival of the next one, is not feasible. In fact, most of the current implementation leverage the parallel architecture of GPUs to concurrently process multiple independent samples and reduce the training times. This is, of course, not feasible for the case of real-time online learning, since each data sample must be processed before the next one arrives. Collecting samples into a batch and processing them all together is not a suitable solution, as it dramatically increases the response time for earlier samples. Current research often focuses on Data Parallelism, which is an extension of the above mentioned paradigm, in which independent data is concurrently executed by different GPUs, and Model Parallelism, where a neural model is split into sub-models which operate concurrently on independent parts of the computation graph and eventually reconstruct a single output. A particular Model Parallel technique which is gaining traction for its general purpose practicality is Pipeline Parallelism, which is actually an Asynchronous Model Parallel paradigm that straightforwardly applies to feed-forward neural networks (Huang et al., 2018). Nevertheless, these approaches are still focused at batch-mode learning, applying the Pipeline Parallelism over mini-batches with independent data traveling through the pipeline. There is currently little research on how to apply these types of parallelism on streams of individually available samples.

To summarize, we see some open problems in the task of learning from Video Streams in a Continual and Lifelong Learning setting. a) current approaches are still far from the idea of learning agents that live in a dynamic environment, relying on static environments and episodic benchmarks, b) there is the need of suitable environments in which to let learning agents freely live; c) there is little research on how reliable and robust are virtual environments to malicious individuals that want to corrupt benchmarks and training; d) there are no parallel computing paradigm explicitly designed for online real-time learning. In the next section, we introduce the contributions in this directions that will be presented in the rest of the thesis.

1.2 Contributions

This thesis aims at answering part of the open problems discussed in the previous section, designing and developing engineering solutions while evaluating their potential impact on enabling new research that would be otherwise considered too hard or costly, allowing researchers to break free from the common assumption of

having huge datasets on which to perform offline batch-mode training, allowing instead the real-time online generation and processing of new data, ultimately enabling research on Continual and Lifelong Learning in a real-time online setting.

The first contribution of the thesis is SAILenv, a 3D Virtual Environment developed in the Siena Artificial Intelligence Laboratory, explicitly designed for the generation of visual streams for enabling task relative to visual recognition, with support for easy extension of the available scenarios even by users with low expertise on Computer Graphics, since this lack of expertise is the most common reason for renouncing to use powerful tools such as 3D physical simulators. The contribution is exposed with more details in chapter 3, focusing on the design and ease of interfacing with Machine Learning frameworks. SAILenv is able to generate multi-modal visual streams, producing in particular photo-realistic visual streams with associated metadata (i. e. dense pixel-wise segmentation, motion information, depth, etc.), with particular attention to efficiency and performance to achieve real-time generation and communication capabilities and ease of interfacing to users' code. SAILenv is carefully designed to allow the creation of agents that may freely live, move and experience the available scenarios. The design of SAILenv pushes towards the direction of enabling Continual and Lifelong learning tasks in real-time online settings, with real-time generation of data and low-latency communication to common Machine Learning frameworks, with extensible scenarios that are easily crafted to the needs of Continual Learning tasks. *Based on (Meloni et al., 2021a).*

In chapter 4, the thesis focuses on the second contribution, that is a parametric framework that allows to dynamically generate replicable scenarios in which to let the agent live, with moving objects and the possibility of slowly introducing new objects and information to enable tasks that are usually framed as class-incremental settings. We describe the theoretical groundings of the parametric framework, proceeding to describe the implementation and integration into SAILenv and the use of these tools to generate growingly complex scenarios through some examples, and how such scenarios can be used to enable new research. The parametric generation framework pushes in the direction of Continual Learning, allowing the researcher to generate scenarios that gradually increase in complexity, allowing the measurement of stability-plasticity capabilities of the learning agent without having to resort to episodic benchmarks that might make the experimental setting too artificial. *Based on (Meloni et al., 2022a).*

The third contribution, presented in chapter 5, focuses on the study of the reliability of Virtual Environments to Adversarial Attacks performed on 3D objects. In the chapter, we describe a possible scenario in which an attacker has access to easily available differentiable renderers with which to perform adversarial attacks on publicly available objects from the Virtual Environment library. To do so, we design a variation of the Projected Gradient Descent attack that takes into consideration the

fact that we are attacking a 3D object that could be seen from various viewpoints and the fact that the Virtual Environment does not produce the same outputs as the differentiable renderers, to devise a saliency-based attack that focuses on salient pixels of the object inside the Virtual Environment. We evaluate how easy it is to transfer these attacks even with limited resources, demonstrating the importance of discussing how to process contributions for Virtual Environments to keep them reliable and trustworthy tools, suitable to be used as the environment on which learning Agents can live and learn useful information that might be harmlessly used in the real-world. *Based on (Meloni et al., 2021b).*

The fourth contribution, deals with the open problem of not having proper tools for accelerating the processing of real-time online visual streams. In chapter 6, we introduce PARTIME, a multi-GPU parallelization library to wrap around neural models to accelerate them in the context of real-time online learning. The contribution takes inspiration from the recent works in Pipeline Parallelism paradigm for batch-mode processing, designing and implementing a variation that focuses instead on accelerating training from samples received one at a time from a temporally coherent visual stream, as it would be the case in online real-time Lifelong Learning, achieving considerable speed-ups that scale almost linearly with the number of used GPUs, at the cost of an approximated gradient computation that is shown to be acceptable in the context of smoothly evolving visual streams, which would be the primary case of application for learning and living agents that might be used in real-world scenarios. *Based on (Meloni et al., 2022b).*

To summarize, the contributions of the thesis are the following,

- Design and implementation of SAILenv, a 3D Virtual Environment specifically designed for real-time generation of fully annotated visual streams and enabling living learning agents;
- Description of a theoretical parametric framework for generation of dynamical visual scenes suitable for Continual Learning, among other tasks, as well as the design and implementation with SAILenv alongside examples and case studies, such as class-incremental tasks in which objects are slowly introduced increasing the scene complexity;
- Study on the feasibility of transferable Adversarial Attacks on Virtual Environments, evaluating how easy it is to transfer such attacks to Virtual Environment and show the need of careful administration of public contributions;
- Design and implementation of PARTIME, a multi-GPU parallelization library to wrap around neural models to accelerate them, applying the Pipeline Parallelism paradigm shifting it from the context of batch-mode offline learning to that of real-time online learning.

1.3 Structure of the Thesis

This thesis is structured as follows.

- In chapter 2, we briefly summarize the current state-of-the-art in the fields of Virtual Environments, Continual Learning, Adversarial Attacks, and Parallelism.
- In chapter 3, we describe in great detail the design and the implementation of SAILenv, describing its structure, the design choices, how to easily interface it with Machine Learning frameworks for carrying experiments in the field of visual recognition and learning agents freely living and experiencing the Virtual Environment. We also present experimental results to prove its photorealism and its usefulness as a benchmark tool for visual recognition tasks.
- In chapter 4, we describe a theoretical parametrical framework for generation of dynamical visual scenes that can more appropriately resemble the approach of Continual and Lifelong learning, with new information and objects that are gradually and incrementally added to the simulation, increasing the complexity of the scenario and allowing the measure of the capability of the agent to learn without forgetting previously acquired knowledge.
- In chapter 5, we study the feasibility of transferable Adversarial Attacks from novel tools that enable 3D adversarial attacks to Virtual Environments, we argue the potential of malicious users to poison benchmarks created through the use of Virtual Environments, proving the possible implications of data poisoning with a set of experiments designed with this particular question in mind, introducing a PGD saliency based attack that takes into consideration the particularities of the Virtual Environment graphical renderer.
- In chapter 6, we describe PARTIME, a Python library specifically designed for easily wrapping neural models and enabling multi-GPU acceleration for the task of real-time online Continual Learning, often times not tractable with modern neural architectures due to their computation-intensive nature that prevent them from having short enough response times to process a visual stream sample-by-sample, that is finishing the processing of a sample before the arrival of a new one.
- Finally, in chapter 7 we draw conclusions on the presented work, expressing further open questions and presenting some hypothesis on possibly interesting future work.

Chapter 2

Background

In this chapter we define the technical terms, the state of the art, and the main concepts involved in the description of the engineering solutions used across the thesis. In section 2.1 we briefly describe the technical details of Virtual Environments, introducing the terms that will be used in chapter 3 and in chapter 4. In section 2.2 we introduce the concepts of Continual and Lifelong Learning, briefly describing some of the solutions for Continual Learning tasks at the current state-of-the-art and introducing the open problems that will be addressed in the thesis. In section 2.3 we introduce the terms related to rendering, useful for understanding chapter 3 and chapter 5. In section 2.4 we introduce Adversarial Attacks, a taxonomy to categorize them and a formalization of the type of attacks that will be described in chapter 5. Finally, in section 2.5 we introduce parallelism techniques for Machine Learning, with a brief description of Pipeline Parallelism, concepts that will be used again in chapter 6.

2.1 Virtual Environments

In the recent decades, the Machine Learning research community started to show an increasing interest in 3D physical simulators as a mean to artificially recreate experimental settings close to real world settings but at a fraction of the cost. These kind of physical simulators are generally known as Virtual Environments, or 3D Synthetic Environments, 3D Simulation Platforms, but for the rest of the thesis we use the term Virtual Environment to indicate them. During the recent years, Virtual Environments have been improved and evolved, steadily increasing the capabilities of physical and rendering engines, leveraging the industrial standards commonly used for game development and re-purposing them as scientific tools for Machine Learning. Virtual Environments have been used in recent years to enable an array of experiments that would have been too costly to set-up in a real-world setting. Formally, we can define a Virtual Environment as a software that allows the user to:

choose a *Scenario* (or even build a custom one) populated with photo-realistic objects meant to recreate a real-world setting; place an entity inside the scenario, called *Agent*, that can experience the world in an egocentric manner through the simulation of different type of sensors (i. e. RGB camera, depth sensors, audio sensors) and augmented with metadata useful for training (i. e. semantic segmentation, optical flow, instance segmentation); define dynamics and behaviors for the objects in the scene, allowing interactions and moving objects; communicate the experiences of the Agents to other software libraries to generate datasets or to provide a real-time stream of the environment through the eye of the Agent. This capability is crucial when considering the huge costs incurred by researchers when trying to collect and annotate datasets that are big enough to properly train recent neural models with their ever growing complexity. Virtual Environments prove their importance even more so when considering learning agents that continuously live, learn and interact with the environment.

Virtual Environments rely on 3 main components:

- A physical engine, which should replicate with high fidelity the physics of the real world, while allowing the customization of physical behaviors to introduce object dynamics for the purposes of experiments. Some Virtual Environments employ custom made engines, but currently the most advanced ones, in terms of degree of realism, use industrial standard engines such as Unity3D.
- A rendering engine, that is a piece of software that takes as input the description of the scenario and the 3D objects inside of it and produces a 2D image with the attached metadata and sensor data. The rendering engine is customizable up to a certain extent, allowing the definition of Agents with different kind of real-world setups in terms of cameras, field of view, noise levels, etc. In section 2.3 renderers are described in more detail.
- A communication protocol, which interfaces the Virtual Environment to a Machine Learning framework through the proxy of the Agent. Communication protocols are designed to have low overheads in order to maximize the bandwidth and minimize the response times of the data incoming to the learning algorithm, and concurrently guarantee the fidelity of the data.

Synthetic data can in general be a powerful aid in this matter. In particular, with Virtual Environments, researchers can easily and procedurally generate potentially unlimited amounts of annotated data, with costs in both time and money that are incomparably smaller with regards to those incurred with collecting and annotating the same amount of real data. Furthermore, for many tasks such as automated driving or automatic control of a robotic arm, reinforcement learning is a commonly used solution (Rao, 2000): to enable reinforcement learning in these situations, re-

alistic and controllable simulations are essential for the training of the model.

Dataset Generation One of the issues that the research community had to tackle once the complexity of architectures rose to unprecedented levels was how to train such a huge number of parameters. In some cases, Big Data allowed the creation of equally huge datasets that made possible to train the networks with outstanding results. This was the case of ImageNet (Deng et al., 2009; Russakovsky et al., 2015), Microsoft COCO (Lin et al., 2014), and others. Unfortunately, this is not the case in many real-world applications, where data collection is hard, or costly, or simply unfeasible. In such cases, the most common go-to solution was Transfer Learning and Domain Adaptation, using state-of-the-art models that performed well in similar context and adapting them with low amounts of data on the desired task. While this could in principle provide an acceptable performance, it is still sub-optimal. The research community then started to consider 3D computer graphics to create synthetic data that could act as surrogate for big data, while using expensive real data in small amounts for ensuring that the network would be able to pick up details missing in the 3D renderings but useful in real-world applications.

During the last decade, many synthetic dataset were generated and made available to the research community. The encompassed tasks are varied, including but not limited to Object Detection (Meloni, 2019; Di Benedetto et al., 2019, 2021), Automated Driving (Johnson-Roberson et al., 2016; Quiter and Ernst, 2018), Pose Estimation (Fabbri et al., 2018). We summarize some of these works to highlight the successes of synthetic data in being a surrogate to real-data and still achieving high performances in many tasks from Object Detection to Automated Driving.

We can consider the work in (Meloni, 2019; Di Benedetto et al., 2019, 2021) to analyze a case where real data collection is often hard and why Virtual Environments can solve many of the issues involved in annotation and collection. In these works we study the case of protective personal equipment detection in construction sites, a task with significant practical applications but with critical issues in the data collection phase. In fact, to the best of our knowledge, public datasets for that task did not exist previous to the referenced works. The reasons of such difficulty, beside the need of manually annotating camera feeds, are all straightforward: data collection activities require an agreement between the researchers and the construction company; workers too must agree to privacy policies so that their pictures can be used, alternatively images must be properly anonymized; it is likely that workers from the same construction company all use similar equipment, reducing the generalization capability of models trained on such dataset; collecting data in such a way limits the number of negative examples (i. e. images where workers do not use equipment) since it is not legal to operate in dangerous conditions; finally, camera feeds collected in such a way are sparse, in the sense that only a few frames

will actually contain worker activity, and they will have to be manually inspected to remove useless frames. All of these problems are solved in the referenced works with the use of Virtual Environments, in particular the graphics engine of the GTA V videogame. It is used to generate a suitable synthetic dataset, on different construction sites, with variable equipment, with no need for privacy policies, plenty of examples with workers in dangerous conditions, and dense videos without idle times.

Given these compelling reasons, an interesting problem left is whether a model trained on synthetic data is able to generalize well enough to be used on real applications, or if it will learn particular patterns that belong exclusively to the Virtual Environments. In (Johnson-Roberson et al., 2016), authors trained an automated driving model on 50K and 200K synthetic images, and the resulting network outperformed a network trained on around 3K real images when tested on a validation set made of real images. Their work proved a positive result of synthetic data, that can in specific settings completely replace real data.

Similarly, in (Meloni, 2019; Di Benedetto et al., 2019, 2021) the main positive result is that synthetic data can be used in conjunction to a smaller real dataset to significantly improve the generalization capability of the network. However, the accuracy on the real test set is around 76%. One of the factors is that the Virtual Environment did not allow to closely replicate the domain distribution of the real test set, since most of the images not in the test set are not taken from security cameras. This highlights an interesting requirement for the Virtual Environments: while in some cases it is not necessary for the synthetic data to be perfectly photo-realistic (Mayer et al., 2018), the environment must be carefully handcrafted to match the test domain distribution. While this may seem a time-consuming task, it is important to remind that it is a one-time only job that will later allow to create virtually infinite data.

Regarding the photo-realism quality of the synthetic data, (Mayer et al., 2018) proposes interesting guidelines: for low-level computer vision, such as optical flow, realism is often not necessary, as shown with the Flying Chairs dataset (Mayer et al., 2016); it is instead useful (but not necessary) simulating flaws of real cameras, such as distortion and blur; finally, it helps taking advantage of the generative capability of Virtual Environments to create different virtual datasets with varying domains, such as lighting and weather conditions. Conversely, high-level computer vision tasks benefit of high photo-realism data, reducing the gap that have to be bridged when performing domain adaptation before the deployment and use on real data.

Summarizing, Virtual Environments are powerful yet flexible tools that allow the research community to reduce costs in the generation of the huge amounts of data necessary for training current large models, and have proven themselves in many occasions to yield accurate and precise models even when photo-realism is not as

strong as most recent 3D computer graphics toolkits allow. Of course, Dataset Generation was clearly the most obvious and immediate use of Virtual Environments, given that it reflects a common assumption in Deep Learning, that is training with batch-mode offline procedures. While this is of course an important aspect of Machine Learning that drove the evolution of the Artificial Intelligence field until now, we argue that this is not the most natural paradigm of learning, and it is certainly different from the way humans learn. We will argue in more detail about this statement in the next section.

In the recent years there has been interesting work in this direction. We can take as an example ALFRED (Shridhar et al., 2020), a benchmark for navigation and instructions interpretation generated through the AI2-THOR environment (Kolve et al., 2017), which contains a great amount of episodes and high-level instructions that can be used to train an agent to interpret language directives and an egocentric vision to perform a sequence of tasks in an indoor environment. Another example is AGENT (Shu et al., 2021), a benchmark for core psychological reasoning procedurally generated with ThreeDWorld (Gan et al., 2020), containing a significant amount of episodes that can be used to test four types of reasoning (goal preferences, action efficiency, unobserved constraints, and cost-reward trade-offs) and probe key concepts of core psychological reasoning.

Real-time streams Humans, and more in general animals, live inside a dynamical environment, where visual stimuli change in a continuous way and follow strict constraints given by the laws of physics. This paradigm is significantly different from the idea of huge datasets that collect big amounts of loosely correlated data. In fact, one could easily notice that a child does not need annotations for every pixel in their retina, but it only needs few supervisions on objects on which they are focusing their attention (Betti et al., 2022). There is certainly value in studying learning agents that live and learn in an open environment, making them experience the world in which they operate similarly to how humans do. In fact, this goes in the direction of Lifelong Learning. Lifelong Learning is an advanced paradigm that involves an agent that learns continuously, accumulating knowledge from the past, adapting it to new stimuli and putting it to use to effectively learn new knowledge and solve future problems (Chen and Liu, 2018). It differentiates itself from the paradigm of learning from huge datasets as it involves open environments, as opposed to closed environments. It necessarily needs to be semi-supervised, relying on few supervisions from other agents (initially humans, but potentially even other more knowledgeable agents), while acting on self-supervisions when faced with information that can be interpreted through the lens of past knowledge.

Once again, Virtual Environments are clearly a good fit for tackling the problem of letting a learning agent live in an open environment. They eliminate the risks

of letting an embodied agent learn in the real world, potentially posing dangers to humans and things around them while they first learn how to navigate and interact with the world; they reduce the costs of prototyping, since with Virtual Environments the cost of repeating an experiment is merely launching the software again, while in a real environment it would involve rebuilding or at least reprogramming an embodied agent; supervisions can be provided by humans, but also by the Virtual Environment itself: in fact, by definition, the Virtual Environment software knows what is showing to the agent, and can therefore automatically respond to queries by the agent on the nature of objects.

In summary, Virtual Environments can easily enable this paradigm with few tweaks and adaptations to the task. In fact, the current technology in 3D physical simulation already allows real-time performances in the simulation of open worlds with thousands of simultaneous interactions between entities and users. They also fancy highly photo-realistic qualities: it is the case of state-of-the-art engine such as Unity3D, which powers many of the videogames that are played all around the world on consumer-grade hardware. The future of this field is paved by the recent release of Unreal Engine 5, which employs ray-tracing technologies (see section 2.3) to allow the creation of visual streams so realistic that an untrained eye cannot distinguish it from a real video taken by a real camera. For example, see Figure 2.1. The same technological solutions used for generating video datasets such as AGENT (Shu et al., 2021) and ALFRED (Shridhar et al., 2020) can be readily extended to enable the paradigm shift and go through Lifelong Learning. This direction was taken for example in (Marullo et al., 2022), where a Virtual Environment was used to create a visual stream on which an agent is trained to perform Optical Flow estimation in a Continual Learning fashion.

In this direction, this thesis covers the design and implementation of SAILenv (Meloni et al., 2021a), which was intended to enable the generation of fully annotated real-time photo-realistic streams through the use of Unity3D engine. The platform was in fact used to enable Continual Learning research in (Tiezzi et al., 2022b), where visual streams generated in real-time by SAILenv were used as visual stimuli for the unsupervised learning of visual features, and the annotations in the stream were used for a few-shot open-class and class-incremental learning of Object Segmentation task. SAILenv is covered in more details in chapter 3 and chapter 4.

Safety concerns It is clear that Virtual Environments can be a powerful tool for many tasks and research work. Another important quality of Virtual Environments is that they can allow contributions and collaboration between research groups from different backgrounds. An interesting example is the possibility of contributing to the Environment by sharing objects and scenes produced for personal experimen-



Figure 2.1: Rendering demo with Unreal Engine 5, showcasing the photo-realistic qualities of the engine with a video so realistic that an untrained eye cannot distinguish from a real video. Image from <https://www.unrealengine.com/en-US/tech-blog/environment-artist-explains-how-he-created-near-photo-realistic-train-station-using-ue5>.

tation, enlarging the library of readily available object and making it easier for researchers with less experience in Computer Graphics to work on algorithm research without being hindered by the data generation task. Researchers more versed in physical simulations and programming can help by extending behaviors and dynamics available in the simulator, helping other researchers with less experience in physics. Therefore, we argue that an open-source approach to the distribution of these tools would certainly help the tool to prosper in the research community. Such is the approach taken by SAILenv and AI2-THOR, for example.

There is unfortunately another side to the coin, that is the fact that if we consider the possibility of letting a loosely regulated community to contribute to the projects, we also must consider that there may be malicious individuals that intend to poison benchmarks generated through Virtual Environments. A malicious individual could pose as a regular contributor to the object library, proposing objects that appear as regular objects to moderators of the community when inspected by human eyes, but instead act as adversarial examples to some Machine Learning models (see section 2.4 for more details on Adversarial Attacks). Adversarial Attacks on 3D objects have been proven possible in many different works (Yao et al., 2020; Toheed et al., 2022), but in chapter 5 we show that Adversarial Attacks from easily available differentiable renderers can be effectively transferred to Virtual Environments.

Therefore, research communities that intend to collaborate to maintain a crowd-sourced Virtual Environment must carefully take into consideration the possibility of Adversarial Attacks towards the Virtual Environment and study proper policies to prevent attacks. On a more positive note, the use of Adversarial 3D Objects could be intentionally exploited to study the robustness of neural models to Adversarial Attacks or even use them as training data to increase such robustness (Athalye et al., 2018).

2.2 Continual and Lifelong Learning

The idea of computational systems that can operate in real-world settings comes with the assumption that they will be exposed to streams of potentially multi-modal sensory information from a dynamical environment, therefore they must be able to learn multiple tasks from a data distribution which is, by its very definition, not static. A learning agent living in an environment needs to have a learning process comprising the capability of gradually and progressively increasing and fine-tuning its knowledge, use it for bootstrapping the learning of new tasks (i. e. transfer learning) while avoiding catastrophic forgetting of previously learnt information (McCloskey and Cohen, 1989; McClelland et al., 1995; French, 1999). The ability of learning over time in a continuous settings by adding new tasks to the capabilities of the agent all the while keeping skills learnt from previous experiences is commonly called *Continual Learning*, or even *Lifelong Learning* in the case of agents that never stop learning and thus do not have a clear separation between training and testing phases. During the past decades, this task has been considered a demanding challenge for neural networks in particular, since they are particularly susceptible to the catastrophic forgetting phenomenon, and for the progress of artificial intelligence in general (Thrun and Mitchell, 1995; Hassabis et al., 2017).

Nonetheless, there has been a recent increase in attention directed to Lifelong Learning, mostly for the implications in the development of autonomous robots that show similar capabilities to humans in learning and fine-tuning their skills during their whole lives, extracting information from multi-modal streams acquired through different sensory systems (Calvert et al., 2004; Bremner et al., 2012; Tani, 2016). One of the most distinguishing aspect of lifelong learning in humans and animals is, indeed, the fact that they are born and immediately placed in a dynamical environment, with important interactions and dependencies between the various senses with which they experience the world, which inherently offers a rich set of patterns and regularities that can shape their learning mind (Lewkowicz, 2014; Murray et al., 2016). This complex interactions between the environment and the inherent drive of the living organism to survive and prosper gives the infant organism the ability of self-imposing goals and objectives, further driving the interaction with

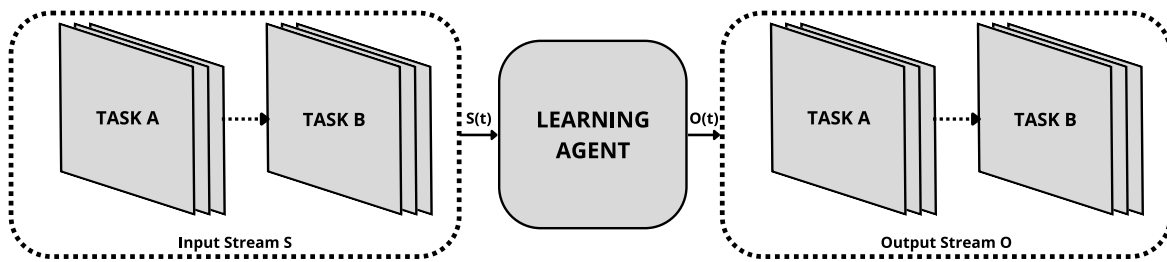


Figure 2.2: Continual Learning schema, where the agent is exposed to a continuous input stream $S(t)$ with various tasks (*Task A* and *Task B* in the example) and is supposed to produce a continuous output stream $O(t)$, autonomously adapting to the changes in the task and in the distribution of data in the input stream.

the surrounding entities and therefore its learning (Gopnik et al., 1999; Cangelosi and Schlesinger, 2018).

However, the approach followed by most research on Continual Learning re-uses concepts from the traditional Machine Learning approaches, devising solutions that incrementally adapt to new tasks but still learn with shuffled and isolated samples. This is significantly different from what humans and animals experience in the real-world environment in which we all live (Cangelosi and Schlesinger, 2018; Krueger and Dayan, 2009). In general, it seems that the approach taken for visual recognition tasks has removed the thread of time from the visual stream that is commonly experienced by humans, shuffling the data in order to simplify the problem, thus losing the patterns and regularities that can be found on temporal sequences of data, focusing only on those related to spatial information. Thus, the immediate consequence was focusing on collection of images instead of video streams, which is actually a problem with many unnecessary degrees of complexity with respect to what biological learners experience (Betti et al., 2022).

As mentioned, one of the most prominent issues with neural networks is the phenomenon of catastrophic forgetting, which happens when training a model with new data brings a sharp decrease in performance or even complete overwriting of past information. An insight into catastrophic forgetting can be given by the consideration that common learning procedures dictate that the entire dataset is available at training phase. Whenever the data distribution changes, the network should be retrained on a new dataset sampled from the new distribution. While this paradigm has been shown to be extremely useful for solving a wide array of tasks (LeCun et al., 2015; Guo et al., 2016), it is not suitable for sequentially learning new tasks through data samples that are progressively made available. In fact, the performance of conventional neural network model is not maintained on previous tasks while progressing in the learning procedure (Kemker et al., 2018; Maltoni and Lomonaco, 2019).

A naive solution like retraining from scratch with all available data could help in maintaining high performances even in previous tasks, but it would clearly not be a feasible solution when considering settings with agents that have reduced memory and cannot store all previous data or agents that must be continuously operating and cannot allow a re-training phase during which they do not work. Another setting in which retraining is not feasible is the setting where the agent needs to operate in real-time, responding to environmental changes with low response times while concurrently adapting to new data distributions without stuttering or blocking for re-training (Cangelosi and Schlesinger, 2018). We expect that a Continual Learning agent is instead able to acquire new information through incoming input streams, use it as a basis and transfer it to newer data, all the while keeping a stable performance on older tasks. This is commonly known as stability-plasticity dilemma and has been the topic of many studies (Grossberg, 1982, 2013; Mermillod et al., 2013; Ditzler et al., 2015).

The research community tackled the problem with various solutions, trying to avoid catastrophic forgetting while avoiding re-training from scratch on incremental datasets. Continual Learning approaches can be separated into three big categories (De Lange et al., 2021): *Replay Methods*, *Regularization-based Methods*, and *Parameter Isolation Methods*. *Replay Methods* approximates the idea of re-training the model on the whole datasets, loosening the requirement by storing only a subset of the past training samples, which are then injected into the training inputs for a new task to refresh the performances of the model on previously seen tasks (Rebuffi et al., 2017; Rolnick et al., 2019; Isele and Cosgun, 2018; Chaudhry et al., 2019). This category of approaches comes with significant limitations: first of all, there is an important memory overhead due to the storage of previously seen samples; furthermore, storing data can be seen as a privacy issue in some tasks, for instance in medical applications, where retaining raw input data is not allowed for legal reasons. There has been research exploring the idea of leveraging generative models to learn the data distribution of inputs from previous tasks and automatically generate suitable inputs to maintain the performances on past tasks, reducing the memory overhead involved in storing used samples and avoiding privacy issues (Shin et al., 2017). *Regularization-based Methods* act on the stability-plasticity trade-off, avoiding to store information about previous tasks, by adding a regularization term to the loss function that drives the learning of the model. For instance, some methods estimate the importance of individual weights, penalizing changes on weights that are considered important for previous tasks (Kirkpatrick et al., 2017; Nguyen et al., 2017; Zenke et al., 2017; Aljundi et al., 2019). Finally, *Parameter Isolation Methods* also act on the stability-plasticity trade-off by isolating the parameters used for each task. Usually, they divide a model into sub-parts, each dedicated to a single task. Usually, these kinds of methods need to be told what task are they currently exe-

cuting, activating the correct branch of the model and limiting the versatility of the solution (Fernando et al., 2017; Mallya and Lazebnik, 2018; Serra et al., 2018; Rusu et al., 2016; Xu and Zhu, 2018).

All the above solutions have been an important step in the direction of incremental learning tasks, but they are still often based on the assumption of randomly shuffled annotated data samples, which is different from the idea of Continual and Lifelong Learning that sees an agent not only incrementing its knowledge on particular domain-specific skills, but actually learning general knowledge and how to apply it to previously unseen situations and tasks. Learning agents that live and operate in the real world are supposed to experience a continuous real-time stream, with multi-sensory information that must be processed efficiently while learning different task concurrently and without interference between tasks (Parisi et al., 2019). Some ideas have been presented in this direction (Betti et al., 2022) which rely on the availability of continuous streams of information. This kind of stream is currently not easily available to researchers, but Virtual Environments are promising candidates for yielding the type of information needed to validate these ideas (Meloni et al., 2021a, 2022a). Similarly, there is currently little research on the parallelization of Machine Learning models to achieve real-time performances on online streams while keeping high accuracies, instead focusing on smaller models that reduce the computational cost to achieve lower latencies, but sacrificing learning capabilities. In this case, further expanding the idea of Pipeline Parallelism for the context of online real-time learning is a promising candidate to solve the issue of deep neural models high response times and low frame-rates (Meloni et al., 2022b).

2.3 Renderers

Since the 1970s, the field of Computer Graphics became increasingly sophisticated, leading to advanced works on rendering techniques and making it a more distinct subject of study. We define rendering as the process that outputs a 2D image of a 3D scene by taking as input the definition of the scene made of 3D objects and their physical properties, the lights that hit the objects, and the properties of the virtual camera on which to project the scene. Formally, we can define the rendering process as a function r that takes a 3D scene s and a camera c to produce a 2D image $I_{s,c}$.

$$I_{s,c} = r(s, c), \quad (2.1)$$

The definition of s is not straightforward and heavily depends on how r is defined. In fact, rendering is a complex process, it is not uniquely defined and the final results heavily depend on the assumptions and physical approximations of the behavior of light that are used in the algorithms. In general, an object is defined by its geometric properties and its surface properties, usually called *surface material*. Thus,

there are different variants of rendering, which are differentiated by the way they represent the geometric properties of objects, namely *mesh-based*, *voxel-based*, *point cloud-based*, and *neural implicit representation-based* rendering, on which recent works are focusing. Similarly, there are variants differentiated by the approximations on the behavior of light, namely *non-PBR*, *PBR*, and *Raytracing*. Commonly, virtual environments and videogames rely on mesh-based rendering alongside PBR and thus we will describe it with more detail. We will instead give a brief account of the other types to allow comparison.

First, let us focus on the differences between the representation of 3D objects. There are three main categories of object model representation for renderers: *Voxel-based*, *Point cloud-based*, *Neural implicit representation-based*, and *Mesh-based rendering*.

Voxel-based rendering. It assumes that the 3D space is discretized into unit cubes, and each of the cubes is assigned a N-dimensional vector that encodes various information about the occupancy of that voxel. Basically, a voxel is the 3D extension of a pixel in a 2D image. Voxel-based rendering is often used in medical imaging, for example in Computed Tomography scans, in which case the tensor is actually a scalar value that contains the opacity to x-rays of the substance in that voxel; in the case of image rendering, the vector contains the description of the surface material of the object in that voxel.

Point cloud-based rendering. It represents object as a list of points defined by their position in cartesian coordinates, attached to the vector describing the surface material. Point clouds can be used in many cases where voxel-based rendering can be used, such as medical imaging, and in other industrial cases such as industrial Computed Tomography to check for differences between a manufactured part and the corresponding project.

Neural implicit representation-based rendering. It assumes that the N-dimensional vector representing the physical properties of a point (x, y, z) is described by the output of a Neural Network $F(x, y, z)$. This method of rendering works similarly to point cloud and voxel-based rendering, but an important difference is that its memory footprint is not coupled to spatial resolution. Since the object properties are encoded by the continuous function F , the object can be rendered at any spatial resolution without needing huge amounts of memory.

Mesh-based rendering. It assumes that the objects are represented by a set of vertices in 3D cartesian coordinates and a set of faces (usually triangles) connecting the vertices. It is widely used in videogames and simulations due to the fact that can represent complex 3D structures in a compact and memory efficient way. Such rep-

resentation is called *mesh*. The surface material is usually represented by 2D images, called *textures*, and each vertex in mesh is mapped to a coordinate on the texture (commonly called *UV coordinates*). Each visible triangle, i. e. the triangle closest to the camera viewpoint and not occluded by any other triangle, is projected on the *camera view plane*. Then, the algorithm determines what pixels are encompassed by the triangle. The properties of inner pixels are computed as a weighted sum of the properties of the three vertices based on the distance from them.

Then, let us focus on the differences between the approximations of light behavior. There are three main categories of light models for renderers: *Non-PBR*, *PBR*, and *Raytracing*.

Non-PBR. It assumes a very simple physical model where the texture represents the color that is refracted by a given point of the mesh. The light information is combined with this information to compute simple shadows and to properly modulate the refracted color based on the color and intensity of the incident light. Often, the quality of the appearance of the object is handcrafted by artists to give a satisfying result from a given set of viewpoints.

PBR. It assumes a more complex physical model which simulates many properties of the interaction between the surface and the incident light. Usually, more than one texture is associated to a PBR material. The main textures are *albedo*, which represents the refraction color of the surface, *normal map* which represents the direction of the surface normal and is useful to represent bumps or imperfections in the surface, *smoothness/metallic* which represents how smooth or rough the object is, or alternatively how metallic-looking it is, and *emission* which represents light emitted directly from the surface. These textures are combined with the properties of the incident light, such as angle of incidence, color and intensity, to produce a convincing and realistic appearance from virtually any viewpoint.

Raytracing. It is very similar to PBR, with the main difference that in this case, ray of lights are individually simulated and the textures and other information about the surface are used to change the trajectory of the ray alongside its color and intensity. This allows Raytracing to give an even more convincing and photo-realistic appearance to the rendering, even more so when considering reflections in bodies of water or metallic objects (these kind of reflections are harder in PBR) and shadows.

As we can see, rendering is a complex process that has no straightforward differentiation, therefore the integration into Machine Learning algorithm is not easily achieved. Therefore, the research community took an interest in developing dif-

ferentiable alternatives for rendering. The first straightforward method was to approximate the gradients, while keeping the rendering process the same (Loper and Black, 2014; Kato et al., 2018; Kato and Harada, 2019; Genova et al., 2018). Other research work focused on making the rendering process differentiable by loosening some parts of the algorithm introducing differentiable approximations (Rhodin et al., 2015; Liu et al., 2019; Chen et al., 2019). A few Differentiable Rendering libraries stemmed from these research works, namely Kaolin (Jatavallabhula et al., 2019), PyTorch3D (Ravi et al., 2020) and Mitsuba 2 (Nimier-David et al., 2019). Differentiable Rendering enabled many interesting research works, such as 3D object reconstruction (Kato et al., 2018; Yan et al., 2016), material estimation (Azinovic et al., 2019), and adversarial attacks (Zeng et al., 2019; Xiao et al., 2019; Liu et al., 2018; Alcorn et al., 2019; Meloni et al., 2021b).

2.4 Adversarial Attacks

During the years, Neural Networks, and in particular Deep Neural Networks have demonstrated a remarkable capacity of learning various tasks with outstanding performances, ranging from image classification to object detection, from pose estimation to autonomous driving. Given their successes, Deep Learning solutions have been chosen to power many services that are used each day by millions of users, such as Cloud AI Computing from Google ¹ and NVIDIA ², and tools such as autonomous cars, malware detection, drones, robotics, face recognition, voice assistants, and uncountably more. The widespread use of these tools have brought forth many advantages to the people, but when this kind of tools are moved from controlled laboratory environments, in which they are studied and designed, to real-world settings, their integrity, safety, and security mustd be guaranteed, since the lack of any of those poses serious concerns to the public interest.

In fact, it is now well known that malicious attackers can easily craft adversarial examples that induces serious mistakes in an otherwise well-behaving Neural Model. It was first shown in the field of computer vision, by applying human-imperceptible perturbations to well-recognized images which would then be classified with an unrelated label (Szegedy et al., 2013; Biggio et al., 2013; Biggio and Roli, 2018) (see Figure 2.3 for an example). Some tentative explanation was given to this phenomenon, suggesting that it was at least partly caused by the significant linearity of neural models, given that many of their components (ReLU, LSTMs, etc.) are designed to behave in a linear way, and non-linear components such as sigmoids are optimized to operate in their mostly linear region (Goodfellow et al., 2014). Also the insufficient regularization of the pure supervised learning approach could be par-

¹<https://cloud.google.com/products/machine-learning>

²<https://www.nvidia.com/object/gpu-cloud-computing.html>

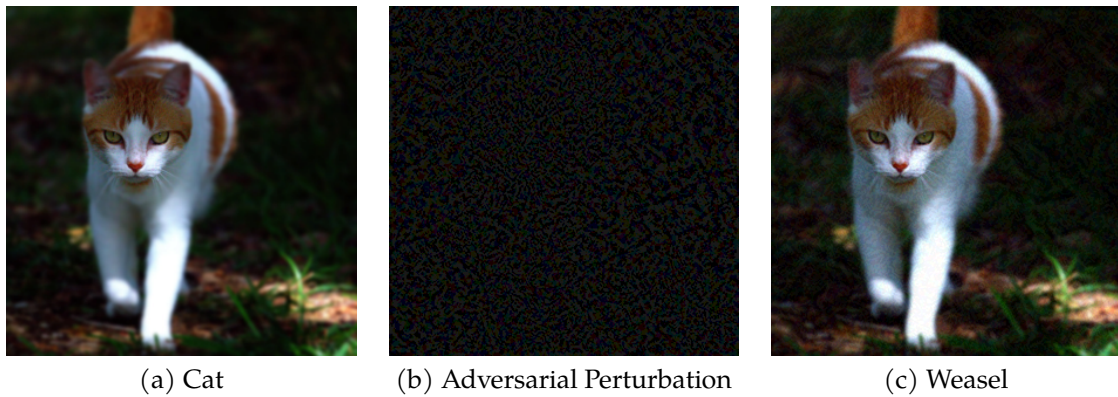


Figure 2.3: Traditional Neural Networks are particularly susceptible to simple Adversarial Examples, where an Attacker can easily craft an imperceptible perturbation (2.3b) to a previously recognized image (2.3a) to obtain a completely wrong classification (2.3c).

tially responsible for the vulnerability to simple perturbations. The susceptibility of neural networks to this kind of attacks was demonstrated even on speech recognition (Carlini et al., 2016; Zhang et al., 2017), and on autonomous vehicles (Kurakin et al., 2016).

Adversarial Attacks can be categorized along various axes through a taxonomy that was first introduced for general Machine Learning (Barreno et al., 2010) and extended later due to new developments in the field of security in Machine Learning (Kumar and Mehta, 2017; Chakraborty et al., 2018), with particular consideration for Computer Vision (Akhtar and Mian, 2018).

In the following, we define terms that will be used in the rest of the thesis to describe entities and concepts in Adversarial Attacks. We define an *Attacker* as an entity that has interest in disrupting the correct functionalities of a system. Usually, attacks to any computational systems are defined by a *Threat Model*, which describes what are the Attacker’s *Goals* and *Capabilities*. It is common for the Attacker’s Goal to either attack *Integrity*, where the intent is evading detection causing *False Negatives* (e. g. preventing the recognition of a cat inside a picture), or *Attack Availability*, where the intent is making the system not available to users by saturating the normal operations through many *False Positives* (e. g. causing recognition of a cat in many pictures to reduce the trust of the user in the system).

The scope of the Attack can be distinguished between *Targeted Attacks*, where the Attacker intends to disrupt the functioning of the model only for a particular class of inputs, while leaving intact the rest of the operations for the rest of the classes, and *Indiscriminate Attacks*, where the Attacker intends to completely deny the functioning of the system for any class. Usually, Evasion Attacks are Targeted Attacks, since the intent of evading detection for an input can be framed as disrupting the func-

tionality for a class of inputs it (it can be even composed of a single example), while Denial Attacks are Indiscriminate Attacks, since they generally cause malfunctioning to the system without focusing on any particular class.

Similarly, we can distinguish the Attacker’s Capabilities based on what part of the model it has access to. We call *White-Box Attacks* those in which the Attacker has complete information on the architecture of the model, and optionally the training procedure or the training set. We call *Black-Box Attacks* those in which the Attacker has limited information on the model and can only query the system by providing inputs and inspecting corresponding outputs. Usually, White-Box Attacks are easier as they can exploit much more information and a wider *Attack Surface*, that is the portion of a model that can be used to freely interact with and to attack the system. An interesting property of Adversarial Attacks is *Transferability*, that is the capability of an attack crafted using a surrogate White-Box model to be effective when attacking a target Black-Box model on which the Attacker has limited knowledge.

For the scope of the thesis, we formalize an Evansion Attack to a Neural Classifier. Let us consider, for instance, a classification task and a generic annotated pair (x, y) , where $x \in \mathbb{R}^d$ denotes an input pattern and y is the associated supervision. We also consider a neural network classifier $\mathcal{C}(\cdot|\theta)$ with parameters $\theta \in \mathbb{R}^p$. Let us indicate with \bar{y} the output yielded by the classifier when processing x , that is $\bar{y} = \mathcal{C}(x|\theta)$, and the loss function $L(\bar{y}, y, x)$ that measures the mismatch between the prediction and the ground truth. A common learning procedure aims at identifying the model parameters θ which minimize the empirical risk function $\mathbb{E}_{(x_k, y_k) \sim \mathcal{X}} [L(\hat{y}_k, y_k | x_k)]$. It has been shown that neural classifiers are vulnerable to the injection of a *perturbation* in the input, which results in the misclassification of the provided input. In particular, we define a perturbation δ that when added to the input x causes the classifier $\mathcal{C}(\cdot|\theta)$ to misclassify the input, i. e. $\hat{y} = \mathcal{C}(x + \delta|\theta)$ with $\hat{y} \neq y$. Of course, stealthy attacks are those that limit the perturbation δ to a set of admissible perturbations \mathcal{P} that prevents human observers from detecting the perturbation while causing the intended damage to the classification. A common choice for this set, in particular for Computer Vision tasks, is restricting the perturbation to fall upon a ℓ_2 -ball or a ℓ_∞ -ball. In the example in Figure 2.3, we can identify x in Figure 2.3a, δ in Figure 2.3b, and $x + \delta$ in Figure 2.3c. In the simplest case, the attackers aims to increase the prediction loss value by chosing a suitable δ , practically finding a solution to the following optimization problem,

$$\max_{\delta \in \mathcal{P}} L(\bar{y}, y, x + \delta). \quad (2.2)$$

There are many variations of attack procedures for solving Equation 2.2 to estimate the best δ for a given task, which usually differ by the properties described in the previous paragraphs, based on the Attacker’s capabilities and knowledge of the model. In the case of CComputer Vision and in particular for the task of Im-

age Classification, for the scope of the thesis, we consider two attack procedures: *Fast Gradient Sign Method*, or *FGSM* (Goodfellow et al., 2014) and *Projected Gradient Descent*, or *PGD* (Madry et al., 2017).

FGSM. This attack was motivated by the idea that the linear behavior of many components in neural networks would make them susceptible to linear adversarial perturbations (Goodfellow et al., 2014). The Fast Gradient Sign Method is based on a linearization of the loss function around the current values of θ , which is then used to compute a perturbation in a ℓ_∞ -ball with radius ε through the following equation,

$$\hat{x} = x + \varepsilon \cdot \text{sign}((\nabla_x L)(\bar{y}, y, x)). \quad (2.3)$$

The attack is very simple, it works with Black-Box Attacks (excluding the loss function, which in the case of classification can often times be safely guessed to be the cross-entropy) and with only one query to the model for each input that the Attacker wants to perturbate. Even in its simplicity, it is extremely effective for many neural models. GoogLeNet, for example, can drop to an accuracy of 0.1% by crafting Adversarial Examples through FGSM (Goodfellow et al., 2014).

PGD. This method can be seen as an iterative version of the FGSM attack. In fact, we see that the perturbation is gradually improved by computing a new linearization of the loss function around the current values of θ and x^t (which implicitly contains the perturbation δ^t), similarly to how gradient descent is performed for training. The Adversarial Example is obtained through the following iterative computation, repeated a given number of times.

$$x^{k+1} = \Pi_{\mathcal{P}}(x^k + \alpha \cdot \text{sign}((\nabla_x L)(\bar{y}, y, x^k))) \quad (2.4)$$

Where \mathcal{P} is the ℓ_∞ -ball with radius ε and $x^0 = 0$ or randomly initialized within the ℓ_∞ -ball. There are strong empirical reasons to consider PGD a “universal” first-order attack, meaning that, while it is not guaranteed that PGD will find the absolute maximum of the optimized function, it will often reach local maxima that have similarly suitable loss values, thus attacks performed through PGD are quite likely to succeed (Madry et al., 2017).

The attacks defined thus far are very effective at altering an input so that it makes a neural model misbehave. However, the perturbation is effective exclusively for that particular input. Further altering the example by some legal transformation, such as rotation or translation, will make the perturbation ineffective most of the times. This is clearly an important factor when considering Attacks towards systems that operate in real-world settings, such as Continual Learning models. A slight change in viewpoint, or a rotation of an object, can easily make a perturbation computed

through the above methods ineffective. While this could seem a good property for systems that operate on real-time visual streams, it is actually possible to craft Adversarial Examples that are effective over a range of transformations, using the *Expectation Over Transformation* algorithm, or *EOT* (Athalye et al., 2018). Instead of computing the gradient of the loss on a single input, EOT leverages a distribution of legal transformations T from which transformations t are sampled and then applied over the input x . Instead of constraining the distance between \hat{x} and x , with EOT we constrain the distance between $t(\hat{x})$ and $t(x)$. This is especially important when the output of the transformation and the original input exist in different domains, as is the case for rendering functions (see section 2.3). In that case, x would be a texture, while $t(x)$ would be the actual rendering of the object under some viewpoint, lighting conditions, and other properties of the scene. Thus, we want the renderings to be similar to each other, not only the textures. In practice, for this example, the optimized problem seen in Equation 2.2 is transformed in the following form,

$$\max_{\delta \in \mathcal{P}_{\varepsilon'}} \mathbb{E}_{t \sim \mathcal{T}} [L(\bar{y}, y, t(x + \delta))], \quad (2.5)$$

with $\mathcal{P}_{\varepsilon'}$ defined as the set of perturbations that satisfies the following constraint,

$$\mathbb{E}_{t \sim \mathcal{T}} [d(t(x + \delta), t(x))] < \varepsilon', \quad (2.6)$$

where $d(\cdot, \cdot)$ is a suitable distance function and $\varepsilon' > 0$. This attack has been successful in crafting real-world 3D objects that could reliably induce error over a broad range of viewpoints in otherwise well-performing classifiers (Athalye et al., 2018), and as will be shown in chapter 5, the algorithm is effective also for objects in Virtual Environments even when using different engines for rendering the Adversarial 3D Object.

2.5 Parallelism

Deep Learning research has seen an ongoing increase in model complexity. It has been shown that scaling up network capacity is often an effective approach to enhance the performances of neural models. On the other hand, hardware capabilities are unable to scale as fast as required by such highly complex architectures, raising the need for the development of alternative parallel computations to take advantage of multiple GPUs.

Parallel computations are usually tailored to the task at hand, by keeping into consideration the characteristics of the algorithm and the available hardware and its topology. This handcrafting process is mostly characterized by a difficult trade-off among flexibility, scaling capacity and achievable performances. We can split parallel computations into two main categories: *data parallelism* and *model parallelism*.

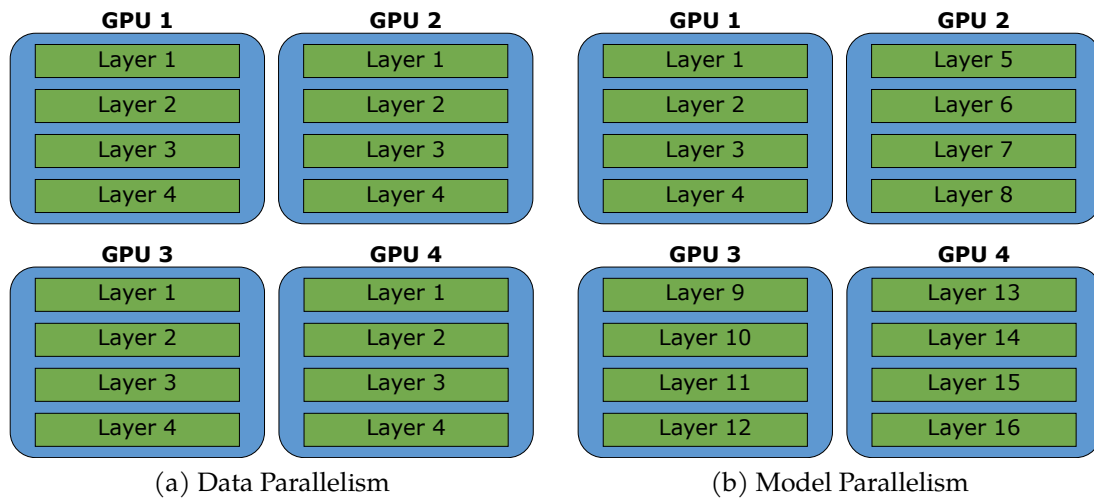
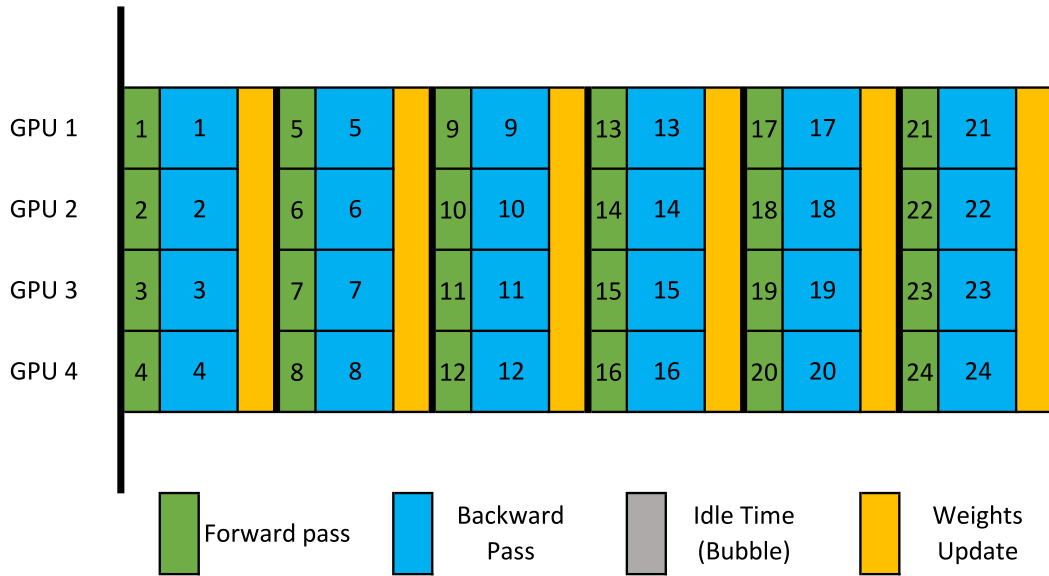


Figure 2.4: Schemes of Data Parallelism and Model Parallelism. With Data Parallelism (Figure 2.4a), a small model with 4 layers is replicated on each GPU, and each replica takes a different input in parallel. With Model Parallelism (Figure 2.4b), a bigger model with 16 layers is split across all GPUs, and each sub-model takes as input the output of the previous sub-model.

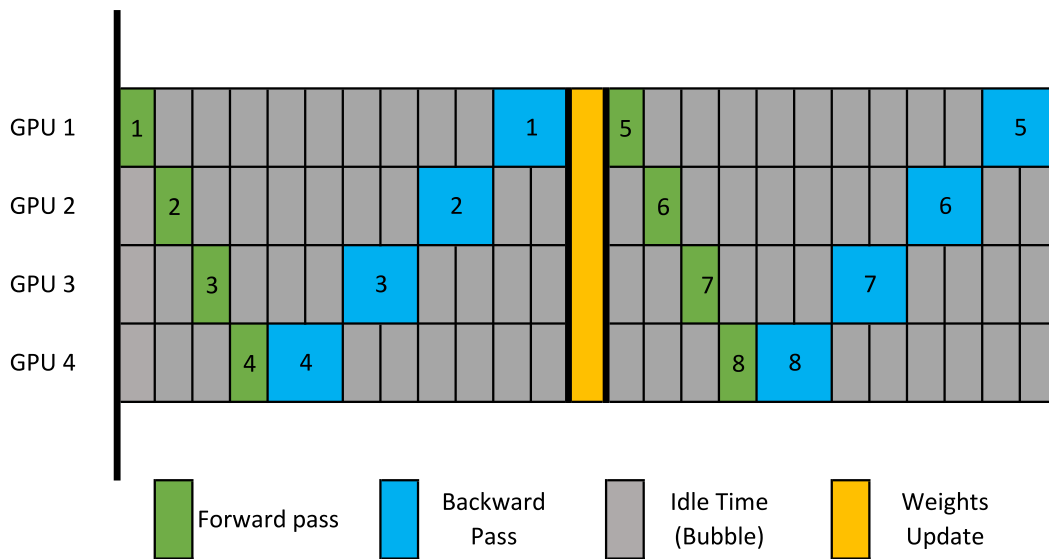
Data Parallelism. The dataset is split into N parts, and each of the parts is distributed to a different GPU. The model is instead replicated into each GPU, which will independently process its own assigned portion of data, aggregating the output with all of the other outputs at the end. Gradients are similarly computed by splitting the correspondent outputs and performing back-propagation on each replica of the model. The benefits of this parallelism paradigm are clear: almost linear speed-up in training and little to no need of adapting the model, which is just replicated and not changed in any way. The weakness is that we can use this paradigm only with models that are small enough to fit in each GPU and it is really useful only when considering batch-mode offline processing. See Figure 2.4a and Figure 2.5a.

Model Parallelism. The model is split into N parts, one for each available GPU, and each of the portions runs on that GPU, the dataset instead is not divided. The main benefits are that portions of the model that work on independent data can run concurrently, obtaining modest speed-ups in some cases, but more importantly it allows to run models that cannot fit into a single GPU. On the contrary, how to split models is not always straightforward and the paradigm often incurs in under-utilization of computational resources. See Figure 2.4b and Figure 2.5b.

It is clear that these two types of parallelism have different cases of applications and they solve two very different problems. In the case of Data Parallelism, we take advantage of multiple accelerators and split the data among them while keeping the



(a) Data Parallelism



(b) Model Parallelism

Figure 2.5: Timelines of execution for Data Parallelism and Model Parallelism. In Figure 2.5b we can see how most of the times each GPU is idle, wasting precious computational resources.

same model, replicated across the GPUs, to significantly reduce the training times, reaching almost linear speed-up with the overhead of synchronizing the replicas after having computed the gradients to keep consistent parameters. In the case of Model Parallelism, we use the memory available to multiple accelerators to split a model that would not otherwise fit into a single GPU, thus enabling it for training without taking into consideration the possible speed-ups of parallel computations. Of course, this is sub-optimal since most GPUs remain idle while waiting for one to complete its computation, therefore another type of Model Parallelism was introduced, that is Asynchronous Model Parallelism.

Asynchronous Model Parallelism. The issue of under-utilization can be tackled with the introduction of asynchronous computations, that is starting processing new data as soon as a part of the model becomes idle. While this clearly reduces idle times, preventing under-utilization and thus significantly speeding-up computations, it also introduces new problems that must be taken into consideration, such as *weight staleness*, i. e. when different data samples use different versions of the weights to derive gradients leading to inconsistent updates and *weight version mismatch*, i. e. when different GPUs have different version of weights at a given instant.

Asynchronous Model Parallelism can include any form of model parallelism which does not synchronize every GPU to a single computational graph, but a straightforward yet effective example of this paradigm is Pipeline Parallelism, which separates a feed-forward computational graph into several non overlapping stages, each one being fed the output of the previous stage, and operating independently as soon as a new input is ready.

Pipeline Parallelism. This paradigm allows to run huge models that would not fit into a single GPU, while at the same time allowing significant speed-ups in the computation. The speed-up is usually heavily limited by the need of synchronizing weights updates and communication overheads between different GPUs. In Figure 2.6 we see the simplest implementation of Pipeline Parallelism as seen in GPipe (Huang et al., 2018), which separately parallelizes the forward and the backward pass, effectively leaving a bubble of idle time which reduces the achievable speed-up, and introducing a flush phase which takes care of issues such as weight staleness and weight version mismatch, but further limiting the possible speed-up. Other types of Pipeline Parallelism have been studied in the recent years, trading off low memory footprint for increased speed-up Narayanan et al. (2019) or gradient accuracy for increased speed-up (Narayanan et al., 2021). In chapter 6 we introduce a Pipeline Parallelism variant, called PARTIME, that keeps limited memory footprint and almost linear speed-up leveraging a gradient approximation, specifically focus-

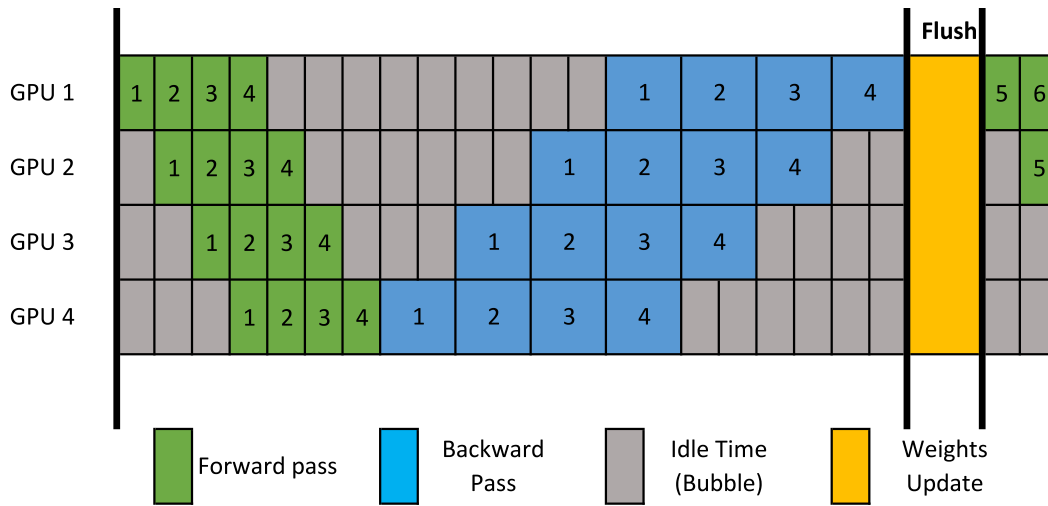


Figure 2.6: Simple Pipeline Parallelism timeline as implemented in GPipe (Huang et al., 2018)

ing on the online real-time processing case.

Chapter 3

Making Virtual Environments Simple: SAIEnv

Developing Machine Learning algorithms to solve a target task usually follows a well-established offline process in which data are collected from the real-world operational environment and then used to learn the models parameters and to evaluate the quality of the trained model. In the recent years, researchers in Machine Learning algorithms, particularly Computer Vision scientists, have shown an increasing interest towards 3D computer simulations of real environments as a mean to artificially recreate experimental settings similar to real world environments (Kolve et al., 2017; Savva et al., 2019; Gan et al., 2020). As mentioned in section 2.1, in the last decade, many different Virtual Environments were released focusing on a diverse array of tasks and situations, such as Automated Driving (Johnson-Roberson et al., 2016), Robotic Arm Control (Rao, 2000), Indoor Navigation (Shridhar et al., 2020; Zhu et al., 2017; Gupta et al., 2017), visual QA (Gordon et al., 2018), and other various tasks (Chaplot et al., 2018; Beattie et al., 2016). These tools allow the researchers to perform controlled tests that would incur in high costs if performed in the real world, both in terms of time and resources, and in some cases dangerous to people and things (Savva et al., 2019).

Most popular benchmarks generated through Virtual Environments and shared by the scientific community have their own characteristic features, with specifically designed 3D environments that accurately resemble the target working conditions.

If we depart from the case of the most popular benchmarks shared by the scientific community, such as the ones aimed at showing the quality of Visual Navigation Algorithms (Shridhar et al., 2020), Visual Recognition (Lomonaco and Maltoni, 2017), and others, each research project has its own characteristic features, and it actually requires to design the 3D environment that correctly resembles the target working conditions. Moreover, the way a virtual agent will exploit the information coming from the virtual world, and how it will react to it, need to be designed

coherently with the target setting. This clearly suggests that there is the need of providing flexible and easy-to-use tools to encourage the use of virtual environments and to favour the development of those research activities that exploit them. Another important consideration to remark is that not all researchers have robust skills in creating 3D scenes, and this aspect might discourage the use of virtual environments.

In this chapter, we present SAILenv, the Siena Artificial Intelligence Laboratory¹, a freely available and open platform² that is specifically designed to cover the needs of a platform to easily design and customize visual environments to experiment with visual recognition and computer vision algorithms in general. The platform comes with an integrated library of photo-realistic 3D objects and scenes, accessible through a light-weight Python library that allows the researchers to quickly prototype experiments interfacing Machine Learning algorithms to the Virtual Environment with few lines of code. SAILenv is based on Unity, a popular game engine developed by Unity Technologies³, that supports many operating systems and includes advanced 3D modeling and state-of-the-art quality real-time rendering. When released, SAILenv was the only platform (to the best of our knowledge) that yielded optical flow data alongside the RGB stream, providing real-time motion-related information inherited from the 3D engine (thus being extremely accurate), and not computed afterwards from multiple 2D observations, as commonly done in optical flow algorithms (Hui et al., 2018; Farnebäck, 2003), providing important data that can be used in benchmarks and training Computer Vision algorithms that benefit from the motion field. Furthermore, differently from other platforms (Kolve et al., 2017; Gan et al., 2020), SAILenv transfers data without relying on higher-level communications protocols (such as HTTP), reducing the communication overhead and allowing higher framerates and lower response times by the Python API.

SAILenv is powered by the Unity 3D engine, which includes a powerful editor to create and customize 3D scenes, that, however, is not always intuitive and might discourage researchers that are starting from scratch on the field of Virtual Environments. To overcome this issue, we designed SAILenv following the approach used by most game development team, that is separating the scene designer by the engine programmer. In facts, Unity allows augmenting the editor with ad-hoc tools that can be programmed and integrated into the existing editor, abstracting many details of the programming interface and allowing unspecialized users to create rich scenes without the need of ever touching the inner code. Thus, SAILenv includes a ready-to-use Unity project, initially derived from AI2-THOR 2.1.0 (Kolve et al., 2017) and later augmented by adding realistic textures and lighting effects by means of state-

¹SAILab, <https://sailab.diism.unisi.it>.

²See <https://sailab.diism.unisi.it/sailenv>

³See <https://unity.com> for further details.

of-the-art texturing software, making them strongly photorealistic. Thus, SAILenv comes with ready-made scenes that can be customized with simple operations of drag-and-drop. Similarly, dynamics can be applied to any object with intuitive tools integrated into the editor, using the included templates or programming new behaviors for expert users. More details are included in section 3.4.

We also provide experimental evidence on the quality of the scenes generated by SAILenv, showing that a state-of-the-art neural model (He et al., 2017) trained on Object Detection with real-world data, can actually easily recognize most of the objects in the SAILenv library, proving its photo-realism, thus suggesting that models trained on SAILenv need reduced adaptation when adapting to real-world domains. Furthermore, we also measure and compare the speed at which SAILenv is able to generate motion features, showing that the platform leads to smaller running times when compared to popular optical flow estimators (Farneback, 2003), including neural models (Hui et al., 2018), apart from intrinsically being more accurate in most cases.

The features provided by SAILenv, ranging from a diverse library of ready-to-use objects and scenes, to a efficient and real-time response oriented data generation and communication protocol, are particularly designed to enable the task of Continual and Lifelong learning from visual streams in a real-time online setting. In fact, the diverse library of objects allows researchers to quickly design scenarios with increasing levels of complexity, gradually incrementing the number of objects in the visual stream and enabling, for instance, tasks as object detection in class-incremental settings, where the agent needs to learn to recognize previously unseen objects without forgetting what they had already learnt. The Python interface through which the environment is accessible by the user code allows the researcher to design experimental protocols without having to manually code complex code for the 3D engine (this will be further explored in chapter 4). Finally, the data generation executes in real-time even on low-end user-grade hardware, while the data communication protocol is handcrafted without high-level dependencies to focus on lowering response times to its minimum, so that the communication does not become a bottleneck that slow down the real-time processing of the visual stream.

The rest of the chapter is organized as follows. In section 3.1 we describe similar platforms and highlight the differences with SAILenv. In section 3.2 we detail the architecture of SAILenv and its design choices. In section 3.3 we describe what kind of data is generated by SAILenv and the structure of such data and associated metadata. In section 3.4 we describe the library of available ready-to-use objects and scenes integrated into SAILenv. In section 3.5 we describe how SAILenv handles dynamical objects and how the movement of the agent is supported. In section 3.6 we describe the Python interface which enables researchers to integrate their Machine Learning code to the Virtual Environment, presenting a few examples that highlight

Table 3.1: Comparison of the main features of SAILenv with other popular platforms. *LightNet* refers to lightweight communication over the network (n.a. means network communication is not directly provided).

PLATFORM	PHOTOREAL	DEPTH	OPTFLOW	LIGHTNET	OS
DeepMind Lab (Beattie et al., 2016)		✓		n.a.	Unix
Habitat (Savva et al., 2019)	✓	✓		n.a.	Unix
AI2-THOR (Kolve et al., 2017)	✓	✓	✓		Unix
ThreeDWorld (Gan et al., 2020)	✓	✓	✓		Win+Unix
SAILenv (Meloni et al., 2021a)	✓	✓	✓	✓	Win+Unix

the ease of use of the API. In section 3.7 we describe the experimental evaluation of the photorealism and generation-communication speeds of SAILenv, implicitly showing how the platform can be integrated into Machine Learning code. Finally, in section 3.8 we summarize the chapter and introduce some works which leverage SAILenv as the platform for generating visual streams.

3.1 Related Platforms

Several environments and simulators have been developed by the scientific community in the last few years. Some simulators are not photorealistic, or they are specifically designed to handle specific tasks. Some examples are DeepMind Lab (Beattie et al., 2016), UETorch (Lerer et al., 2016), Scene (Handa et al., 2016). The main issue with these platforms is that they are not photo-realistic. Moreover, some of them expose the full environment to the agent, while an agent operating in real world does not see the entire environment. For example, a robot that operates in an apartment does not see the entire apartment. Amongst the virtual environments with visual-realistic appearance we mostly focus on the recent AI2-THOR (Kolve et al., 2017) and Habitat (Savva et al., 2019). Other existing frameworks are Home (Brodeur et al., 2017), Chalet (Yan et al., 2018), Gibson (Xia et al., 2018), SceneNet RGBD (McCormac et al., 2017). These environments are used to study embodied agents (Xia et al., 2018), to instantiate tasks that are about visual navigation with reinforcement learning (Zhu et al., 2017; Gupta et al., 2017), interactive VQA (Gordon et al., 2018), task-oriented language grounding (Chaplot et al., 2018) or vision-and-language navigation (Wang et al., 2018).

SAILenv, coherently with what is commonly done in related platforms, captures RGB representations with or without depth information, acquired from the agent camera position and orientation. Similarly to what we propose, also AI2-THOR (Kolve et al., 2017) is based on the Unity engine, but it focuses on the interaction with the environment, so that actions can be attached to objects. Differently, SAILenv fo-

cuses on visual recognition, and it simplifies the assignment of new semantic categories to objects, an operation that does not require knowledge of the code structure, and that can be done through the Unity GUI. Moreover, the client-server architecture of AI2-THOR is based on HTTP communication between Unity and the Python API, where the 3D engine acts as a client while the server is implemented on the Python side of the architecture. SAILenv, as we will describe in section 3.2, implements a more natural organization in which the virtual world is a server to which a Python client connects to retrieve data that will be processed by the target algorithm. Habitat (Savva et al., 2019) is mostly focused in allowing the access to different 3D datasets (Song et al., 2017; Chang et al., 2017) by a uniform interface, and it includes its own fast simulation engine. In principle, the direct customization or creation of 3D environments is possible, but it is not straightforward. For this reason, SAILenv is built around the Unity engine, that is a very popular and multi-platform software solution easily accessible and customizable, widely used for videogames and physical simulations. In Table 3.1 we summarize a comparison of the main features of SAILenv with some of the aforementioned frameworks. The features considered in the table are *Photorealism*, *Depth Rendering*, *Optical Flow*, *Lightweight Network Communications*, and *Operating System*. Each of these features is of particular importance in the context of the generation of Virtual Visual Streams. *Photorealism*, as shown in (Johnson-Roberson et al., 2016; Meloni, 2019; Di Benedetto et al., 2019) and explained in the **Dataset Generation** paragraph of section 2.1, is extremely important to reduce the efforts needed to transfer models between the real and the virtual world. In practice, when deploying a model that was trained on virtual data in the real world, the model needs to be finetuned on real data to account for the differences between virtual and real. The more the virtual data is similar to real world footage, the less effort is needed to achieve higher performances. *Depth Rendering* is useful when generating virtual data to make up for the lack of binocular vision when understanding the depth and relative position of objects in the scene. *Optical Flow* is a fundamental piece of metadata of dynamical scenes, as it encodes how the pixels of each static frame are evolving over time from the agent point of view. *Lightweight Communication* is an important feature of a Virtual Environment that focuses on real-time interactions with a ML agent. Well-known network protocols such as HTTP, which is used in AI2-Thor and ThreeDWorld, are easy to integrate to the Virtual Environment, but introduce overheads which are suitable for dataset generation but not for real-time stream generation, since strict timing are required. Finally, a wide *Operating System Support* is relevant as it allows the researcher to deploy the Virtual Environments on a wider set of systems.

Notice that only SAILenv includes lightweight communication over the network, and that, differently from AI2-THOR (Kolve et al., 2017) and Habitat (Savva et al., 2019), but similarly to ThreeDWorld (Gan et al., 2020) it can also run on a Win-

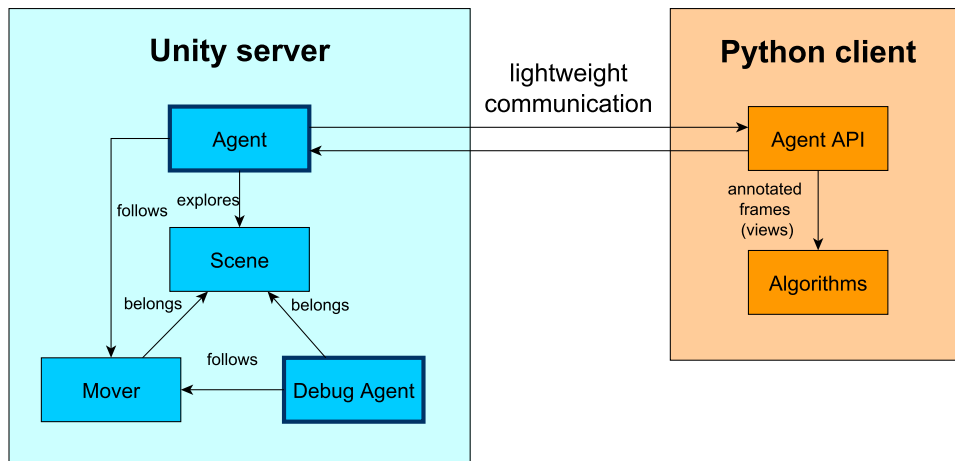


Figure 3.1: Organization of the SAILenv architecture.

dows machine. When SAILenv was released, it was the first to introduce motion information metadata attached to the visual stream, but the feature has been since integrated in AI2-THOR and ThreeDWorld. Currently, ThreeDWorld has the advantage of a wider library of objects, some of them even being fully physically simulated (e.g. foldable cloths, liquids), but the code is not open source and cannot be easily extended. AI2Thor has a procedural scenario generation for indoor exploration which is used for training and testing robotic agents through Reinforcement Learning. Similarly, ThreeDWorld introduced a procedural indoor scene generation, which uses heuristics to automatically create scenes with suitable object density.

3.2 Architecture

SAILenv is organized in a client-server architecture, that in a natural way implements the idea of having a virtual *scene* (simulated in the server) and an *agent* that experiences and explores the environment (controlled by the client). The overall SAILenv architecture is exemplified in Figure 3.1. The agent, which lives inside the Virtual Environment instance, is controlled through the *Agent API*, which consists in a set of high-level commands implemented through a lightweight Python API. The API allows the client to move the agent in the environment, create and delete objects, set their trajectories, and most importantly query the environment for the information experienced by the Agent. When queried, the server replies with a number of views that capture different properties of what the agent is experiencing, that is RGB views with associated metadata, such as pixel-wise labeling, which can be used by Computer vision algorithms, fed to Machine Learning frameworks,

recorded and generally used for any needed purpose.

We built the server within the Unity framework, providing an ad-hoc Unity server that responds to client requests. The Unity server is more than simply a network interface layer; it is a computing module that is in charge of constructing the virtual environment, handling the physics simulation, and real-time rendering while fully utilizing the Unity infrastructure. It generates the data in the views requested by the client, and thanks to the powerful engine embedded in Unity, the physics simulation runs at real-time speed on most laptops and servers, allowing the generation of real-time streams with no hiccups or long wait times between one view and the other. For debug reasons, that will be clearer in the rest of the chapter, the Unity instance running on the server allows to show what the agent camera is currently capturing in the screen attached to the server, allowing some special server-side interactions.

Server and client are connected through a lightweight communication protocol, which ensures minimal overhead for the transmission of data from and to the Virtual Environment. The Unity server awaits for incoming connections on a target port. The client can contact the server through that socket, calling the provided Python API and triggering the generation of the agent inside the 3D environment. The so created agent is associated to a worker thread which runs in background, listening and replying to further requests, but operating in an asynchronous fashion with respect to the physics engine. To an expert Unity user, this choice could seem counter intuitive. Commonly, simulations in Unity follow a simple pattern, in which there is a single thread that runs in a loop (the so called *game loop*), which is the only thread allowed to apply changes and interactions to the simulation. The loop is composed of several phases, which we omit for brevity since it is not fundamental for the understanding of what follows. Each entity in the simulation can register a behavior, in the form of a piece of code or a script, to any phase or even more than one. During each phase, the main thread running the game loop will call every behavior registered to that phase, in no particular order, but synchronously. This means that until a behavior does not return, the main loop will wait indefinitely. If a behavior slows down, all the simulation is slowed. Of course, during network operations, issues such as network slowdowns or communication errors are common, causing the associated agent to slow down. Keeping the network code on a separated background thread allows the simulation to proceed without hiccups due to the network. Since secondary threads cannot directly interact with the simulation, they collect commands from the client APIs and enqueue them on the main thread as soon as they are ready to be executed without further delays. When the command completes execution, the reply is sent back to the secondary thread which sends it back on the network. Finally, the network protocol also offers the possibility of sending raw data or by GZipping them beforehand, reducing the overhead on low-bandwidth networks.

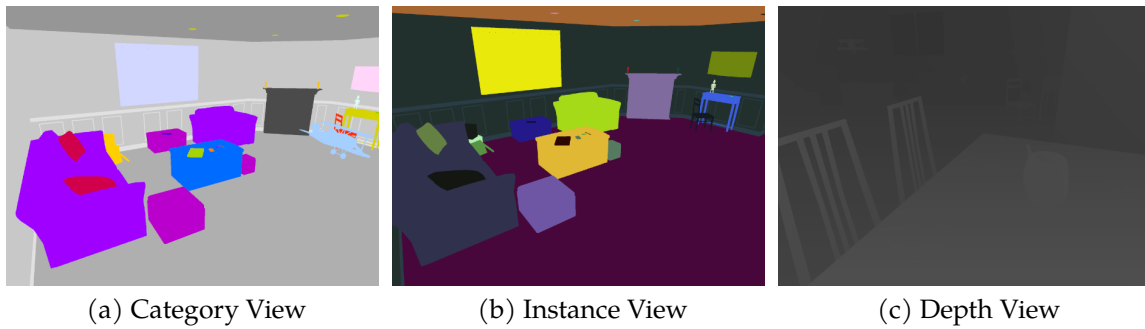
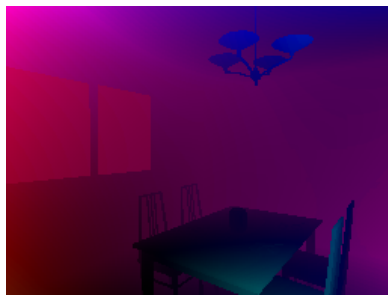


Figure 3.2: Pixel-wise annotations yielded by SAILenv. In Figure 3.2a it is shown the pixel-wise semantic segmentation at category level, that is pixels belonging to the same class of objects (e. g. couch or pillow) have the same label. In Figure 3.2b it is shown the pixel-wise semantic segmentation at instance level, meaning that each pixel has a label that uniquely identifies the particular instance of that object. In Figure 3.2c it is shown the depth view of the scene, where each pixel has a gray-scale value which is whiter for objects close to the camera and darker for objects far from the viewpoint.

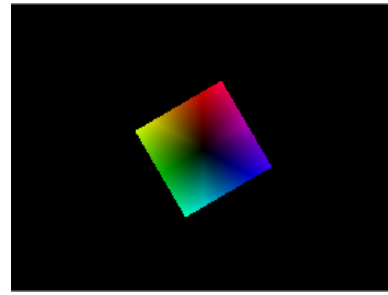
3.3 Details on the generated views

As we briefly mentioned before, the client is implemented as a lightweight cross-platform Python API, with a small dependency tree. It is, in fact, a very tiny interface that exposes high-level commands which allow control on the virtual environment, such as creating a new agent, moving it or obtaining views of the current state of the environment from the point of view of the agent. The views and metadata available through the API are the following: a) RGB view; b) Category Level Semantic Segmentation; c) Instance Level Semantic Segmentation; d) Pixel-wise Depth Labeling; e) Optical Flow. See Figure 3.2 and Figure 3.3 for an example of such views.

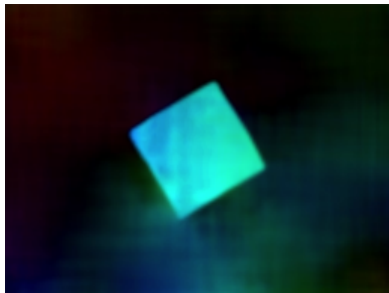
Going into more details, the RGB view is a straightforward representation of what the agent views inside of the environment, representing each pixel with the classic 24-bit encoding; each pixel is annotated with a category identifier (semantic labeling) and an instance identifier, encoded in the category (Figure 3.2a) and instance (Figure 3.2b) views. The instance identifier is automatically and uniquely assigned by Unity when the object is created. The category identifier is instead created and assigned through the Unity Editor, without any code-level operation. In particular, categories are represented as Unity objects (called Scriptable Objects), and they can be attached to every 3D object by a simple drag-and-drop operation. Every scene also includes a *category holder*, which allows the researcher to easily organize set of categories and allow the user to quickly add them to custom scenes. Depth information is taken directly from the rendering engine, represented as gray-scale texture representing the distance of the pixel from the position of the agent



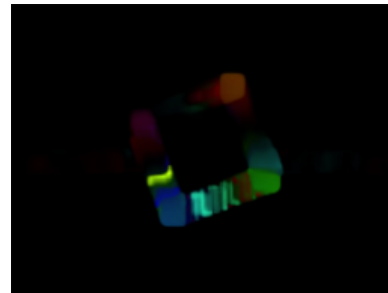
(a) Camera Motion Optical Flow



(b) Rotating Object Optical Flow



(c) LiteFlowNet Optical Flow



(d) OpenCV Optical Flow

Figure 3.3: Optical flow yielded by SAILenv. In Figure 3.3a we see the optical flow given by the agent motion. In Figure 3.3b we see the optical flow given by the objects motion, while the agent stands still. In Figure 3.3c we show how LiteFlowNet (Hui et al., 2018) estimates the optical flow in the same conditions, while in Figure 3.3d we show the estimation of the OpenCV implementation of the Farneback Optical Flow.

(see Figure 3.2c as an example, lighter pixels indicate elements closer to the agent).

SAILenv also yields highly precise and dense motion information about the objects in the environment. Differently from what is commonly done by most optical flow algorithms, SAILenv does not estimate optical flow by observing consecutive views, but is instead fully computed by the physics engine of Unity. Since the engine already holds information about the motion of the objects in the scene in relation to the agent viewpoint (after all, it already uses it to drive the simulation and to render it). This is normally used in games to create visual effects such as motion blur for fast moving objects, but we can easily use the same information to generate a view that includes motion vectors for all pixels of the frame. In detail, such view is a $H \times W \times 2$ tensor of single-precision floating point numbers, being H and W respectively the height and the width of the view. To each of the pixels corresponds a pair of floats which describes the velocity of each pixel in pixels per second. This numerical representation is not easy to interpret and visualize on a screen, therefore SAILenv includes an utility to convert it to the HSV color space, as shown in Figure 3.3. In practice, we consider the pair as cartesian coordinates (x, y) which are

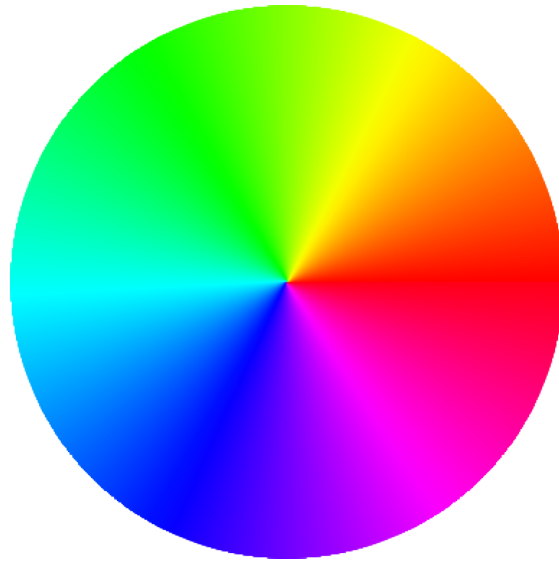


Figure 3.4: HSV color wheel

first converted into polar coordinates (α, Θ) , where α is the magnitude and Θ is the phase. Then, we set $H = \Theta$, $S = 1$ and $V = \alpha$. The resulting HSV image is converted into RGB and shown on the screen. The resulting representation is easier to interpret with the help of the HSV color wheel (Figure 3.4): the value of H indicates the angle in the color wheel (starting from the top), i. e. it associates each color to its respective direction. The intensity of the pixel is instead directly associated to the value V and thus the magnitude of the motion vector: a slow moving pixel will appear darker, a fast moving pixel will appear brighter.

In Figure 3.3b-d, we report three examples of the optical flow computed in a scene that contains exclusively a rotating cube, that has no special textures and a uniform color. Of course this makes it harder to correctly estimate the pixel-level motion using classic algorithms. Nonetheless, SAILenv can correctly represent the rotation of the cube (Figure 3.3b). Instead, widely used solutions such as the OpenCV implementation of Farneback algorithm, or even modern approaches based on convolutional neural networks (Hui et al., 2018), fail to correctly capture the motion, as it is evident in Figure 3.3c and Figure 3.3d. Interestingly, despite its very high precision, SAILenv incurs in an almost null computational burden to compute the optical flow. While some overhead due to data normalization operations and transmission is to be expected, it is still negligible with respect to what is needed to estimate motion from pairs of static frames with other solutions. This will become clearer in section 3.7.

3.4 Photo-realistic Objects and Scenes

We take advantage of the Unity physics engine to handle the virtual environment, allowing SAILenv to rely on all the facilities of the powerful 3D editor integrated in Unity. However, as mentioned before, creating new scenes and objects from scratch is a time-consuming procedure that requires advanced skills in 3D graphics. This is even more clear when working on photo-realistic objects, activity that requires the designer to carefully pay attention to many details and aspects to obtain the target appearance for an object. To help new researchers get into the field and mitigate some of these issues, SAILenv is integrated with a library containing more than 65 objects that can be readily placed in any scene, plus other 3D objects that will help create the structure of new scenes, such as walls, windows, etc. In Figure 3.6 are shown examples of objects in the library. The library is integrated in a ready-to-go Unity project that contains all the photo-realistic elements and four sample scenes, meant to showcase the capabilities of the framework and to run simple prototype experiments in simple context, similar to those described in section 3.7. The user can edit these scenes, adding, moving or removing objects inside it, or create a completely new one from scratch, using the SAILenv library or even adding custom 3D objects.

Sample scenes depict different rooms, with varying contexts, sizes and number of objects. Some of them even include moving objects to evaluate motion-based algorithm. The scenes available are:

- **ROOM 01:** Bedroom. Main objects: laptop, bed, desk, chairs and writing materials. See Figure 3.5a.
- **ROOM 02:** Sitting and dining areas. Main objects: chairs, couches, dining table, paintings. Object movement: a toy rusty plane flies around the room. See Figure 3.5b.
- **ROOM 03:** Bathroom. Main object: toilet, bathtub, cleaning supplies and hands towels. Object movement: many of the objects inside the scene will occasionally be pushed in a random direction, moving from their original position. See Figure 3.5c.
- **OPTICAL:** It includes rotating cubes and a cylinder. This scene is not realistic and is meant to be used for debugging purposes (for example, to test the optical flow feature). See Figure 3.5d.

Most of the 3D meshes are originally from the library of AI2-THOR project, then significantly re-worked, improving their appearance so to reach a more advanced photo-realistic level. To achieve this objective, we employed several state-of-the-art

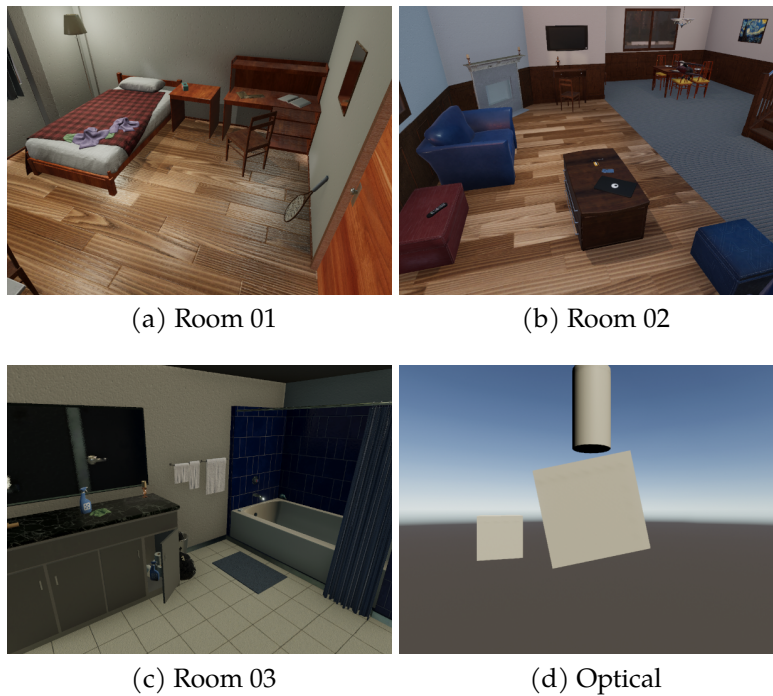


Figure 3.5: Sample scenes in SAILenv

techniques that are commonly used in game design to improve the rendering quality. In particular, we started from *Physically Based Rendering* (PBR), which is a state-of-the-art rendering technique which harnesses advanced physical models that simulates the behavior of light as it comes in contact with the surface of the object⁴. We manually tuned the *Albedo*, *Metallic*, *Specular*, *Emission*, and *Normal* textures to carefully handcraft a believable and realistic effect on rendering. The *Albedo* texture represents the plain color of the texture, regardless of anything else, while its alpha channel represents the opacity; the *Metallic* texture represents how a certain part of a material will act as a metallic surface, from 0 (not metallic) to 1 (fully metallic), while its alpha channel represents the smoothness of the material from 0 (fully rough) to 1 (fully smooth); the *Specular* texture represent how reflective a certain part of a material is, and it is very similar to the metallic one, while its alpha channel works the same as the metallic one (in fact, *Metallic* and *Specular* are exclusive alternatives); the *Emission* texture defines where the material emits light, which can actually illuminate other materials; finally, the *Normal* texture changes the orien-

⁴At the time of writing, a new rendering technique is commonly used on high-end, high-quality projects: Ray-tracing Based Rendering. In Ray-tracing Based Rendering, rays from light sources are individually modeled, allowing seamless rendering of reflections and refractions in a highly realistic manner. During the development of SAILenv, this technology was not available on Unity. Furthermore, the hardware requirements are much stronger, reducing the number of target machines on which SAILenv could be released.

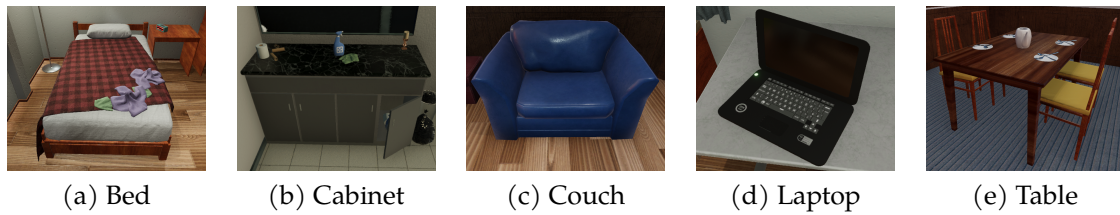


Figure 3.6: Samples of objects available in SAILenv.

tation of the surface normal and is often used to simulate small heights variation without adding polygons to the model mesh.

The basic conditions for global illumination of the environment were built using different HDRI (High Dynamic Range Imaging) skyboxes, which are a set of textures wrapped in cube-maps and applied to the surrounding horizon of a 3D scene. HDRI skyboxes allow realistic environmental global illumination, and static reflections on all objects materials. Each of these skyboxes are based on real-world pictures. Then, static reflection probes and lights were added, generating additional object reflections⁵ and to provide further illumination. SAILenv default set of scenes can be extended by opening and modifying the SAILenv Unity project with the use of the Unity editor. The sample scenes are ready to be edited, otherwise the user can easily create new scenes from scratch. Objects presented in this section can be placed in a scene by a simple drag-and-drop operation from the resources window.

3.5 Dynamical Objects and Moving Agent

Object movement is completely handled by the Unity physics engine, which model objects as a rigid body through the *RigidBody* component. In particular, only objects that have a *RigidBody* component attached to them can move, while objects marked as *full static* are not considered moveable in the simulation. To fully define a rigid body as such, the user must define a mass value that is not supposed to be completely realistic, but appropriate enough to generate a seemingly realistic behavior when applying forces to it. The actual movement behavior can be scripted within the Unity engine, hooking to the physics update phase in the game loop, through code written in the C# language. SAILenv includes three sample movement scripts that can be attached to any object: they were briefly mentioned in the description of the available scenes in section 3.4. The first movement behavior is implemented in the SAILenv script called *POLTERGEIST*, that randomly moves an object by applying

⁵Unfortunately, dynamic objects present a hard limitation as they are not compatible with static reflection probes, meaning that they do not appear in reflections. We are working on improving this aspect, without sacrificing too much performances.

both a force and a torque in a random direction, at random time instants. The second movement behavior is formalized in the `WANDER PLANE` script, that is defined by a configurable set of waypoints; the object will randomly select a waypoint at random time intervals, and the trajectory will result similar to that of a flying airplane wandering around the room. Finally, the last pre-coded included movement behavior is `ROTATE RIGIDBODY`, which simply rotates elements with a configurable angular speed and around a configurable center; it is used in the Optical scene and it is meant as a very simple and useful reference for beginners that want to customize a movement behavior. Integrated tools to enable experiments where dynamical objects and movement are needed are described in chapter 4.

As mentioned above, the Agent too is allowed to move around the scene. When first created, the Python API allows the user to define its starting position and orientation. Since the Python API includes commands to manually set position and orientation, the user could of course develop his trajectory as a Python routine that move the agent accordingly to some custom criteria. To ease the experience of the user, SAILenv scenes also include some immaterial objects that act as a track for the agents to follow and move. An active Agent can be attached to the track, and subject it to the Unity physics engine, acting as a RigidBody for any purpose such as the computation of motion vectors. The consequence is that the movement of an Agent produces motion vectors compatible with the movement of the camera field.

The Agent can also be attached to another RigidBody, making every movement of this other object reflect on the position and orientation of the Agent. This is mostly useful when debugging: in fact, a Debug Agent is created inside the environment at startup, and it is used to show the environment on the screen of the server on which SAILenv is hosted. Enabling the tracking of the Debug Agent by an Agent handled by the Python API, allows the user to manually command the Debug Agent as a proxy of the Agent (and vice-versa). For this purpose, SAILenv includes a `RIGIDBODY CONTROLLER`, that allows the Debug Agent to be guided by means of keyboard and mouse attached to the machine running SAILenv, in a 3D-shooter-game-like fashion: WASD keys are used like directional arrows, while the mouse is used to change orientation. This is particularly useful for debug purposes, as it allows the user to freely explore and see how the algorithm processing the client data will react. Alternatively, the Agents as well as the Debug Agent can be attached to the tracks, mirroring on the screen what the stream received by the client will look like.

3.6 Using SAILenv

The main objective of SAILenv is to offer a simple platform that allows quick prototyping of experiments for visual recognition models with online, real-time streams. In this section, we show examples of how to use SAILenv, both client and server

```
from sailenv.agent import Agent
agent = Agent(width=256, height=192,
              host="192.168.1.3", port=8085)
agent.register()
agent.change_scene(agent.scenes[2])

while True:
    frame = agent.get_frame()

    # run your algorithm using frame data
    # ...
agent.delete()
```

Figure 3.7: Code that starts the Python client and get data from the environment that can be processed by Machine Learning Frameworks.

operations.

The users that want to interface an algorithm with SAILenv, will have to startup the Unity server first, then their Python code will have to import the SAILenv Python library in order to generate a valid client to control an Agent inside the simulation. The Unity server can be started either by using the Unity Editor, that allows to directly run a scene with the embedded Unity server, or by building the project into valid executables for a target operating system. The second option allows for a trivial startup of a Unity server instance and does not depend on the Unity editor, which is available on fewer platforms. After running the executable, the user can select the scene manually from the server screen or remotely from the client API. Once the Unity server is running a scene, the Unity server starts listening for connections on port 8085 (default, but configurable). The Python code needed to instantiate a valid agent and fetch data from its viewpoint is minimal as shown in the snippet of Figure 3.7.

In the code, we assume that the Unity server is running on a server that is reachable by the IP address 192.168.1.3 and is using the default port. Notice that the resolution of the visual stream generated by the Agent viewpoint is configurable during the creation of the Agent. The Agent construction also allows to specify what particular views are required, reducing the computational burden by not rendering and fetching those which are not needed. After having specified the configuration of the Agent, it can be registered to the simulation, triggering the actual creation of the Agent inside the environment. Afterwards, the identifiers of the scenes available to the build are inspectable in the list `agent.scenes`, and can be used to remotely set the running scene by calling the method `agent.change_scene(scene_id)`. The method `agent.get_frame()` fetches the current views of the environment from the viewpoint of the Agent and are returned in a dictionary `frame`. The dictionary is

structured with the following key-value pairs:

- *main*: $H \times W \times 3$ – RGB view.
- *category*: $H \times W$ – semantic labeling, in which each element at coordinates (x, y) of the tensor is an integer containing the category ID associated to the object to which the pixel (x, y) belongs.
- *object*: $H \times W \times 3$ – instance labeling, that is a BGR image in which each pixel color is the unique identifier of the scene object to which the pixel belongs.
- *flow*: $H \times W \times 2$ – optical flow, composed of v_x, v_y velocities of the flow.
- *depth*: $H \times W \times 1$ – the depth of each of the pixels of the agent camera, in $[0, 255]$.

The final call to `agent.delete()` removes the agent from the server, releasing resources. Another important property available to the Agent is `agent.categories`, which is a list populated when the agent registers or when the scene is changed, which contains a dictionary that maps category numeric IDs to their respective names (e. g. pixel with category 24 => *table*). Finally, the position and orientation of the agent can be set respectively by `agent.set_position((x, y, z))` and `agent.set_rotation((rx, ry, rz))`, and the track following behavior is toggled with `agent.toggle_follow()`.

3.7 Experimental Evaluation

We evaluated the concrete quality of SAILenv photo-realism exploiting state-of-the-art object detection neural models, and by comparing the generation speed of optical flow data to other commonly used algorithms.

Regarding photo-realism, we setup an experimental setting in which a powerful object detector, trained on real-world data, returns both bounding boxes and pixel-wise object masks on a stream generated by SAILenv. Focusing on the sample scenes of SAILenv, we measured the capability of such model to correctly recognize object categories on which it was trained, thus measuring how strongly the appearance of SAILenv objects resembles the appearance of equivalent real-world objects. In particular, the model chosen for this measurement was Mask R-CNN based on the popular ResNet-50 backbone (He et al., 2017), pre-trained on COCO-train2017 data (Lin et al., 2014). We focused on a subset of the categories available on the COCO dataset, in particular we chose 14 classes that are shared with the available SAILenv categories. For each object we chose 5 frames generated by SAILenv, in different viewing conditions. The chosen categories are shown in the first column of Table 3.2.

For each class, we measured the average *Intersection over Union* (IoU) between the pixel-wise predictions provided by the Mask R-CNN mask branch and the ground truth category segmentation returned by SAILenv. The resulting accuracies are reported in the second column of Table 3.2, and they show that Mask R-CNN is able to robustly identify a large portion of the objects, despite the different viewing conditions and the virtual settings. In fact, the produced masks are mostly overlapped with the ground truths generated by SAILenv, with some mild exceptions that are mostly due to the difference in the labeling protocol that is followed by COCO training dataset and by SAILenv. As an example of the different labeling protocols, see Figure 3.8. The masks predicted by Mask R-CNN tend to occupy all the area of the object, differently from the highly detailed pixel-wise labels produced by SAILenv, in which the spaces among the structure of objects (e.g. between leaves on a plant or within the net of a racket) are not marked with the object label. In general, we observe a similar behavior in the presence of occlusions or non-dense structures. We explain the reduced performances on book, spoon and fork by noticing that COCO training set presents these objects from very specific viewpoints, while the SAILenv stream shows them with more variety. This observation will be once again important in chapter 5 (see Figure 5.5). To further investigate this intuitions, we also computed a measure that takes account of how strongly the bounding box of the predictions matched the ones of the ground truth, leading to a third column in Table 3.2 (Bounding Box IoU). In this case, we observe an average increase in prediction quality, overcoming some of the aforementioned issues (see, e.g. potted plant and tennis racket). We can also clearly see large standard deviations around some other critical categories (spoon, fork, dining table, chair). This indicates that the recognition quality was high for some viewpoints, while significantly lower from other viewpoints.

Regarding optical flow computations, we remind that SAILenv generates a dense optical flow that is not estimated through the observation of pairs of frames, but is instead inferred by the real motion information coming from the 3D physics engine. Clearly, this leads to the most accurate motion estimation one could have. However, it is not yet clear what is the computational burden of this solution with respect to other competing algorithms, such as the popular OpenCV implementation of the Farneback algorithm (Farneback, 2003), and one model based on convolutional neural network, LiteFlowNet (Hui et al., 2018), one of the fastest solutions available.

The OpenCV implementations takes advantage of other OpenCV tools to speed up the computation, which, in the default Python distribution, is performed using the CPU. Differently, in the case of a PyTorch implementation of LiteFlowNet, GPU-based computations (with the CUDA toolkit) are exploited. For each compared method, we measure the time needed to produce the flow at six increasing target resolutions, reported in the x-axis of Figure 3.9. On the y-axis we report the

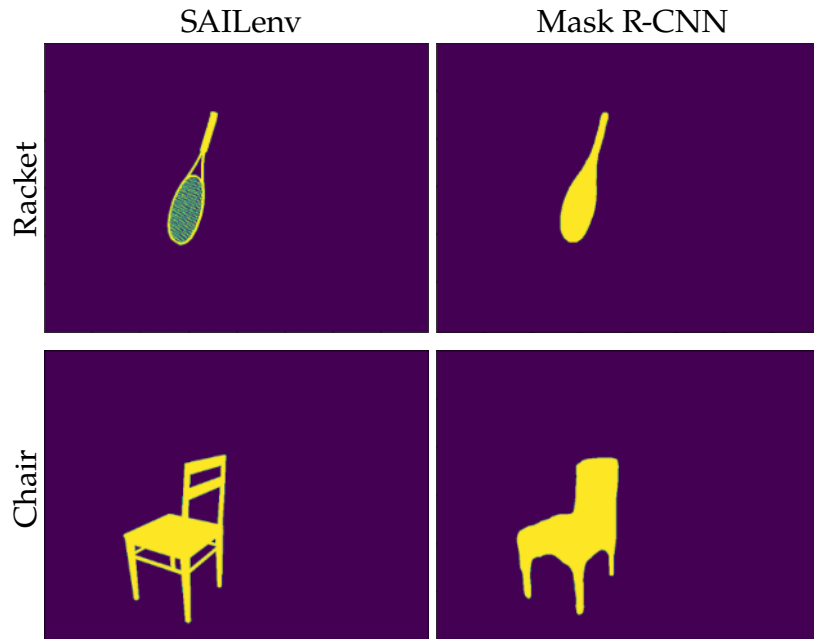


Figure 3.8: Comparison of labeling protocols between SAILenv and Mask R-CNN. As expected, Mask R-CNN learns from the COCO training dataset to densely label even empty spaces in the structure of the objects. SAILenv instead precisely labels pixels belonging to the object, leaving out pixels of the background.

average time over 100 sampled frames, with 95% confidence intervals. In our experimental setting, when measuring the computational speed of the OpenCV implementation and LiteFlowNet, we setup the SAILenv client to only fetch the RGB frame from the virtual environment, turning off all internal optical flow computations and generation of other views. Then, the optical flow is computed using one of the competitors, using the current frame and the one returned at the previous time step. To increase the fairness in the measurement, we take account of the time needed to transfer data to/from the GPU and subtract it from the measurements. For the results shown in Figure 3.9 we used a Windows desktop machine equipped with an Intel Core i9 9900K, 3.60 GHz, 64GB of RAM and an NVIDIA GTX 1080 GPU with 8GB of VRAM. We also performed experiments on two other machines, obtaining results with analogous trend. Clearly, SAILenv takes full advantage of the information from the physical engine, computing precise optical flow and outperforming the competitors in response time. Furthermore, for lower resolutions, it reaches real-time performances even with consumer-grade hardware, thanks to its direct access to the physics engine that takes advantage of heavily optimized GPU-based computations. The gap with other competitors is even more evident at higher resolutions.

Table 3.2: Mean and standard deviation of the predictions of the Mask R-CNN model (pretrained on COCO2017 dataset) on a dataset obtained from the SAILenv sample scenes. Two measures are considered: Pixel-wise IoU and Bounding Box IoU (see the paper text for details).

CATEGORY	PIXEL-WISE IoU	BOUNDING BOX IoU
bed	0.7830 ± 0.0879	0.8201 ± 0.0894
book	0.3347 ± 0.2749	0.3506 ± 0.2870
chair	0.6235 ± 0.0566	0.5557 ± 0.4162
couch	0.8742 ± 0.0533	0.9121 ± 0.0561
dining table	0.6891 ± 0.0398	0.4553 ± 0.4096
fork	0.4599 ± 0.1274	0.4800 ± 0.4294
laptop	0.9551 ± 0.0098	0.9476 ± 0.0207
airplane	0.7193 ± 0.0314	0.7865 ± 0.1005
potted plant	0.6106 ± 0.0499	0.8894 ± 0.0656
remote	0.8980 ± 0.0400	0.9534 ± 0.0127
spoon	0.4036 ± 0.1984	0.3787 ± 0.3611
tennis racket	0.5120 ± 0.0475	0.9548 ± 0.0127
toilet	0.9274 ± 0.0178	0.9623 ± 0.0201
tv	0.9641 ± 0.0171	0.9673 ± 0.0135

3.8 Discussion

SAILenv is a software platform that makes it easy to create, run and get data from realistic 3D virtual environments, on which a researcher can efficiently evaluate visual recognition or other vision-related algorithms, especially in the setting of Continual Learning from visual streams. In fact, with regard to continual learning, a set of extensions were developed and integrated to enable parametric generation of visual scenes, work described in chapter 4, which was then used to generate visual streams that enabled a Continual Learning task (Tiezzi et al., 2022b). The platform was then also used to study the potential impact of malicious actions against the Virtual Environment through adversarial contributions that aim at poisoning benchmarks and scenarios created through SAILenv, study described in chapter 5.

In (Tiezzi et al., 2022b), three visual streams were generated with SAILenv, crafting three virtual scenes where different objects move around the scene in a closed loop, changing orientation, relative size with respect to the observer, and in general showing different viewpoints of the object. The three scenarios portrait contexts of different complexities: the first, `EMPTYSPACE` shows a gray background with a chair, a laptop, a pillow, and a teapot; the second, `SOLID` shows a gray background with three textureless solid shapes, i. e. a sphere, a cylinder, and a cube; finally, `LIVINGROOM` shows a more complete indoor context, with various objects filling the view and possibly creating distractions for the neural model. In this research work, we

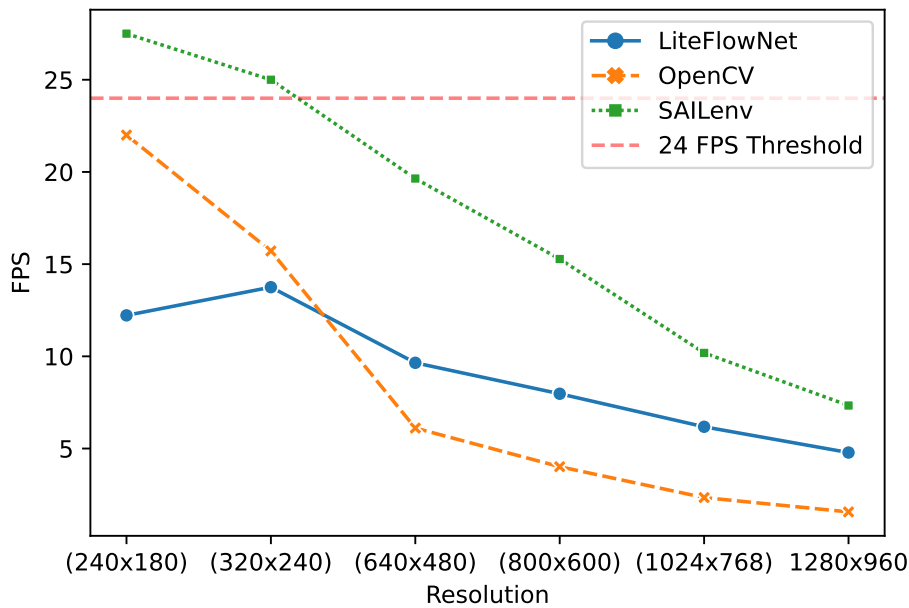


Figure 3.9: Average **Frame Per Seconds** (FPS) achieved when computing the optical flow associated to a frame sampled from the SAILenv scenes. We compare the SAILenv performances with an OpenCV-based implementation of the Farneback algorithm and the neural model LiteFlowNet (Hui et al., 2018).

proposed a novel neural-network-based approach for developing pixel-wise representations in a video stream, progressively and autonomously. The proposed solution is built on a human-like attention mechanism that enables the agent to learn by seeing what object is moving in the attended pixels. Spatial-temporal stochastic coherence along the attention trajectory is then combined with a contrastive term, providing an unsupervised learning criterion that is inherently fit for the considered settings. Unlike other previous research, the learnt representations are used for an open-set class-incremental classification of pixel in the frame, trained with only a few supervisions. The 3D streams aforementioned are used to demonstrate that the proposed solution can learn to differentiate objects simply by watching the video streams, and then recognize them with those few supervisions.

In chapter 4, we describe a theoretical framework for the parametric description of scenes to allow easy generation of novel visual streams with variable levels of complexity, number of objects and object movement dynamics, enabling various and different contexts on which to train and benchmark visual recognition models. To do so, we extended SAILenv, proving the easily extensible design, with a new set of commands and API that implemented the theoretical framework proposed, and showed some examples and possible cases of use of the available tools to create interesting streams for different tasks. *Based on (Meloni et al., 2022a).*

In chapter 5, we study how to craft adversarial 3D objects by slightly altering their textures with the use of a software tool chain appositely designed and implemented with conveniently accessible components. We demonstrate that it is possible, and indeed rather simple, to generate adversarial object using off-the-shelf limited 3D renderers that can compute gradients with respect to the parameters of the rendering process. Furthermore, we show that assaults may be transferred to more complex 3D engines, up to some extent. To achieve this, we propose a saliency-based attack that intersects the two classes of renderers in order to focus the perturbation on texture elements that are anticipated to be most effective on the target 3D engine. Finally, we also analyzed the impact of these attacks on popular neural classifiers. *Based on (Meloni et al., 2021b).*

Chapter 4

Dynamic Virtual Environments for Continual Machine Learning

Traditional machine learning techniques usually assume static input data and the existence of a neat distinction between a training and a test phase. Input data, entirely available at the beginning of the learning procedure, are processed as a whole, iterating over the training dataset multiple times, many epochs, in a batch mode fashion, optimizing the performance with respect to a given learning task. The trained models are then frozen and exploited for inference only, hence computationally expensive re-training procedures are needed to possibly incorporate any new available information. This learning paradigm is clearly incompatible with what humans (and, more in general, animals) do in their everyday life, continuously acquiring and adapting their knowledge to the dynamic environment in which they live. The field of Machine Learning that aims at simulating this learning process by an artificial agent is known as Continual or Lifelong learning (Parisi et al., 2019; Van de Ven and Tolias, 2019). The agent should be malleable enough to integrate new knowledge and, at the same time, stable enough to retain old information. This is known as the *stability-plasticity dilemma* (Abraham and Robins, 2005). Vanilla neural networks have been shown to struggle in this aspect, since training a network to solve a new task will likely override the information stored in its weights, phenomenon known as catastrophic forgetting (McCloskey and Cohen, 1989; McClelland et al., 1995; French, 1999). In the context of Computer Vision, Continual Learning algorithms are trained and their performance assessed on datasets containing static images, such as MNIST (Lecun et al., 1998) or Caltech-UCSD Birds-200 (Wah et al., 2011), or short sequences of temporally coherent frames, such as iCubWorld (Fanello et al., 2013; Pasquale et al., 2015, 2016), and CORE50 (Lomonaco and Maltoni, 2017), usually considering a sequence of distinct learning tasks. However, in our opinion, the resulting learning scenarios are still far away from the original idea of an agent learning from a continuous stream of data in a real-world environment.

See for instance the *task-free* continual learning approach of (Aljundi et al., 2019). Furthermore, having the possibility to fully control the visual scene that the agent perceives (number and types of objects that are present, their pose and their motion, background, possible occlusions, lighting, etc.) is essential to devise a suitable and feasible continual learning protocol and, from this point of view, real-world footages are not a viable alternative.

Taking another significant example, consider a task in which a learning agent has no access to a fully-annotated dataset of sequences of tasks such as the above, but is instead learning from a continuous stream of data where the interactions with the supervisory signals are loosely distributed over space and time. An example of such scenario would be an agent freely living in an interactable environment, with an occasional supervision from another party (a human supervisor, or possibly a second artificial agent with communication skills). In this scenario, the agent would have to independently form its own understanding of the environment around it, and then integrate the external supervisions to give an identity to the learnt features. To do so, the role of time is fundamental to keep a coherence between objects and entities that move around in space, in some sense knitting together otherwise statical images, that is the individual frames of the visual stream. This would be harder considering the kind of datasets and continual learning settings discusses above, since sequences of static datasets miss the time coherence between individual data samples and episodic visual data do not have time coherence between themselves.

This idea is explored in the study conducted in (Tiezzi et al., 2022b), where continuous visual streams respecting such definition of time coherence were used, depicting a virtual scene with various objects moving around the scene, altering their orientation, relative size, and viewpoints. The three scenarios portrayed contexts of differing complexities, ranging from a minimal scene showing a gray background with three textureless solid shapes including a sphere, cylinder, and cube, a more complex scene with the same gray background but with more interesting objects such as a chair, laptop, pillow, and teapot, up to a more realistic third scenario, depicting a complete indoor context with numerous objects filling the view, thereby potentially causing distractions for the neural model. The researchers proposed a novel neural-network-based approach to develop pixel-wise representations in a video stream progressively and autonomously. They utilized a human-like attention mechanism that enabled the agent to learn by identifying which object was moving in the attended pixels. Spatial-temporal stochastic coherence along the attention trajectory was combined with a contrastive term, providing an unsupervised learning criterion suitable for the considered settings. Unlike previous research, the learned representations were utilized for an open-set class-incremental classification of pixels in the frame, trained with only a few supervisions. The results show that by taking advantage of the continuous structure of the stream and by being able of

extracting information from how objects evolve through space over time, few supervisions limited to single pixels in the visual stream are more than enough to train an adequate pixel-wise object detector. Such results highlight the need of scenes that follow the principles of continuous evolution over time, and therefore imply the need of easily creating and customizing such streams to include more scenarios of different complexity to learning agents¹.

We thus propose to exploit the recent technological advancements in 3D virtual environments to create a parametric generator of photo-realistic scenes in a fully controlled setting, easily creating customizable conditions for developing and studying continuous learning agents. As mentioned in the previous chapters, in the last few years, due to the improved quality of the rendered scenes, 3D virtual environments have been increasingly exploited by the machine learning community for different research tasks (Beattie et al., 2016; Gan et al., 2020; Kolve et al., 2017; Savva et al., 2019; Weihs et al., 2020; Xia et al., 2020) and different environments, based on different game engines, have been proposed so far, such as DeepMind Lab (Beattie et al., 2016) (Quake III Arena engine), VR Kitchen (Gao et al., 2019), CARLA (Dosovitskiy et al., 2017) (Unreal Engine 4), AI2Thor (Kolve et al., 2017), CHALET (Yan et al., 2018), VirtualHome (Puig et al., 2018), ThreeDWorld (Gan et al., 2020), SAILenv (Meloni et al., 2021a) (Unity3D game engine), HabitatSim (Savva et al., 2019), iGibson (Xia et al., 2020), SAPIEN (Xiang et al., 2020) (other engines). Moreover, a recent work (Lomonaco et al., 2020) proposed a novel non-stationary 3D benchmark based on the VIZDoom environment to tackle model-free continual reinforcement learning.

Motivated by this significant amount of research activities, we propose to exploit such technologies to implement a method for the generation of synthetic scenes with different levels of complexity, and that depends on well-defined customizable parameters. Each scene includes dynamical elements that can be subject to random changes, making the environment a continuous source of potentially new information for continual learning algorithms. Another key aspect in the context of continual learning is related to the source of supervisions. 3D environments can naturally provide full-frame labeling for the whole stream, since the identity of the involved 3D objects is known in advance. This paves the way to the implementation of active learning procedures, in which the agent asks for supervision at a certain time and coordinates, that the 3D environment can easily provide. Moreover, in the context of semi-supervised learning, it is of course straightforward to instantiate experimental conditions in which, for example, supervisions are only available during the early stages of life of the agent, while the agent is asked to adapt itself in an unsupervised manner when moving towards a new scene. On the other hand, one could also de-

¹We disclose in advance that, indeed, the visual scenes used in (Tiezzi et al., 2022b) were actually created with the methodology described in the rest of the chapter

wise methods where the learning model evolves in an unsupervised manner and the interactions with the supervisor only happen at later stages of development (i.e., for evaluating the developed features). Finally, we introduce the perspective in which scenes could be just part of the same “big” 3D world, and the agent could move from one to another without abrupt interruptions of the input signal.

The rest of the chapter is organized as follows. In section 4.1, the proposed generative framework is described, where visual scenes will be described on a general level and possible factors of variations will be encoded parametrically, allowing the user to fully control the input stream perceived by the continual learning agent. section 4.2 will present a practical implementation of these ideas extending the 3D virtual environment described in chapter 3. Some illustrative examples will be given in section 4.3. Finally, section 4.4 will draw some conclusions.

4.1 Parametric Generation of Environments

This work focusses on the problem of generating customized 3D visual environments to create experimental conditions well suited for learning machines in a *continual learning* scenario. In this section we describe the conceptual framework that allows us to formally introduce the automatic generation of a family of dynamic visual scenes. We would like to underline that one of the main strengths of the automatic generation of 3D environments is the possibility to easily change and adapt them to facilitate the creation of benchmarks with different *degrees of difficulty* with respect to a given model and task, allowing researchers to craft ad-hoc experiments to evaluate specific skills of the continual learning model under study or to design a range of gradually harder learning problems.

The three key factors that we consider in order to devise an automatic generator of dynamic 3D scenes are visual quality, reproducibility and user-control in the generation procedure. When designing the generation of such an environment we have to take into account at least three distinct aspects. First of all, it is important that the visual quality of the rendered scene is good enough to simulate photo-realistic conditions. On the other hand, a flexible generator should not be constrained to such high-level quality and should be able to handle also more elementary scenes in which, for instance, objects are geometric primitives or they have no or poor textures. At the same time, the generating procedure should be easy to reproduce. The dynamics of the scene should be controllable at the point in which it is possible to go back to the very beginning of the agent life to reproduce the exact same visual stream; of course, this does not exclude pseudo-randomic behaviour of the environment as, in that case, the reproducibility can be guaranteed by explicitly fixing the initial condition of the driving pseudo-random process (seed). Scenes with high visual quality and reproducible conditions can readily be obtained as soon as one

relies, for the visual definition and management of the scenes, on a modern graphical engine which is capable of physics simulations, as we will show in our actual implementation in Section 4.2.

Concerning the capability of customizing the generated scenes, the quality of the generator depends on the flexibility it offers in terms of compositional properties and user accessibility to such properties so to effectively describe the generating process of the scenes.

To this aim, we parametrically describe the visual world assuming that we have at our disposal a collection of pre-designed visual scenes $S = \{s_1, \dots, s_n\}$. For each scene s_j , a definite collection of object templates $\Omega_j = \{\omega_{1,j}, \dots, \omega_{n_j,j}\}$ is available, where n_j is the number of object templates in the j -th scene. Each s_j is initially populated by some static instances of the object templates. The parametric generation procedure instantiates new objects from the template list, eventually including multiple instances of the same template (e.g., positioning them in different locations of the 3D space—for example, a table with *four* chairs). Formally, fixing a scene $\sigma \equiv s_j \in S$ with templates $\Omega \equiv \Omega_j$, we can define the collection of N objects that will be added to σ by the parametric generation procedure as $\Phi := (\varphi_1, \dots, \varphi_N) \in \Omega^N$.

Practically, this means that given a scene, there will be a set of object templates, and the scene can be populated with any number N of objects, allowing the same template to be picked several times. For example, given a scene with templates $\Omega = \{\text{chair}, \text{pillow}, \text{laptop}\}$, we could have $\Phi = (\text{chair1}, \text{chair2}, \text{pillow1}, \text{laptop1})$, where $N = 4$ and we used numerical suffixes to differentiate repeated instances of the same object template.

In this work, we assume that the lighting conditions of the rendering engine are fixed and so the position and the orientation of the agent point of view². We are also assuming that the scene σ is populated with an additional set of objects apart from those in Φ , with the latter being the only ones allowed to move in the environment. We denote with $(v_k)_{k \in \mathbb{N}}$ the sequence of frames captured by the agent point of view. Hence, σ can be generated once Φ is chosen and the following attributes are specified for each φ_i :

- the indices $(k_i, \hat{k}_i) \in \mathbb{N}^2$ of the frames where φ_i makes respectively its first and last appearance;
- the position and the orientation of the object in the frame k_i , collectively represented as a vector³ $\pi_i \in \mathbb{R}^6$;

²Here we are making this assumption in order to simplify the management of the generation procedure, however these settings can be regarded as additional parameters that can be chosen to define the environment.

³Again, for the sake of simplicity, we are assuming to work with objects which are rigid bodies (hence the \mathbb{R}^6) but indeed this is by no means a crucial assumption.

- its trajectory (i.e., its position and orientation) for each k such that $k_i < k \leq \hat{k}_i$, modeled by a set of parameters indicated with τ_i and defined in what follows.

Notice that, in order to grant additional flexibility to the scenario definition, it is useful to allow the possibility of dynamically spawning new objects on the fly, when the agent is already living in the generated environment. This property enables the creation of scenes that might also significantly change over time, being expanded or connected to other scenes, capabilities that might be very appropriate in the context of continual learning. The values of (k_i, \hat{k}_i) , π_i , and τ_i , for $i = 1, \dots, N$ are regarded as parameters that characterize the customizable objects visible in a frame k . In particular, parameters τ_i , $i = 1, \dots, N$ unambiguously define the object trajectories, such as the trajectory's global shape, the speed and whether or not the trajectory completely lies in the agent's field of view. Formally, considering the i -th object, we have that $\tau_i = (\kappa_i, \vartheta_i^1, \dots, \vartheta_i^m)$, where κ_i specifies the chosen kind of trajectory while $\vartheta_i^1, \dots, \vartheta_i^m$ stand for all the additional parameters required to fully determine it. Overall, the visual environment is specified by the collection of parameters $\Theta := (k_1, \dots, k_N, \hat{k}_1, \dots, \hat{k}_N, \pi_1, \dots, \pi_N, \tau_1, \dots, \tau_N)$.

Hence it is clear that through the choice of Θ we can control the number of objects present at any given frame k , the position and orientations of the objects, the way in which objects moves and their velocity, i.e., the nature of their trajectories and whether or not objects escape the field of view. A fine control over this set of parameters provide us with a general tool to create highly customizable datasets suitable for continual learning scenarios, possibly of increasing difficulty with respect to a given learning task. For example, in an object recognition problem, the number of angles from which an object is seen, which is closely related to the chosen trajectory, could clearly affect the visual complexity of the task. A fine control over this set of parameters provide us with a general tool to create datasets of increasing difficulty with respect to a given learning task. For example, in an object recognition problem, the number of angles from which an instance of an object is seen, which is closely related to the chosen trajectory, could clearly affect the complexity of the task.

4.2 Continual Learning 3D Virtual Benchmark

SAILenv Meloni et al. (2021a) is a platform specifically designed to ease the creation of customizable 3D environments and their interface with user-defined procedures. With a few lines of code, any learning algorithm can get several data from the virtual world, such as pixel-level annotations. SAILenv includes a Unity library with ready-to-go 3D objects and it provides basic tools to allow the customization of a virtual world within the Unity 3D editor, without the need of writing 3D graphics specific code. Differently from the other existing solutions, it also offers motion informa-

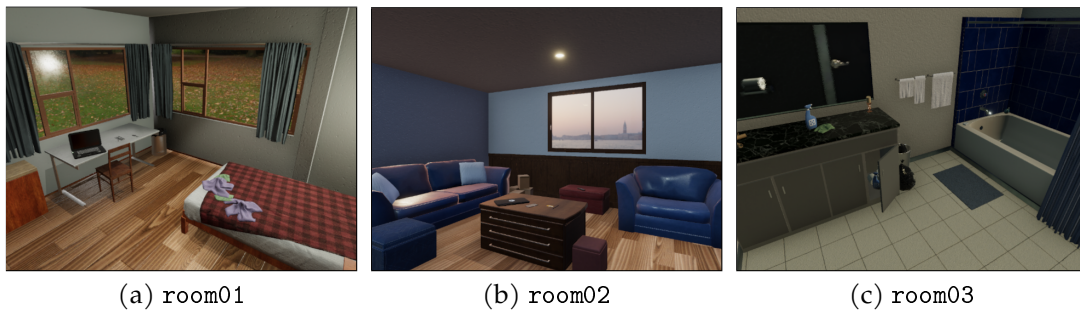


Figure 4.1: The three default scenes of SAILenv, room01, room02, room03 (besides the empty scene `object_view`).

tion for each pixel of the rendered view. SAILenv is based on the Unity Engine⁴, a state-of-the-art graphics and physics engine that is commonly used for videogames and physical simulations. It therefore presents realistic objects and scenes, with fine details and realistic illumination, while allowing the creation of credible motion dynamics of objects in the scene. The SAILenv platform, when executed, creates the virtual world, managing the physical simulation in all its aspects. It also opens up a network connection listener, which waits for incoming connections to interact with the environment. The communication is implemented with low-level socket operations and a custom protocol which focuses on achieving high performance, avoiding bottlenecks in data exchange that would excessively slow down every simulation, for reasons not-related to machine learning.

The platform is released with a Python API, which offers a high-level interface, called *Agent*, that acts as the main player in the communication between the 3D world and custom Python code. The API allows the creation of multiple agents that “live” in the virtual world, each of them with its own view of the environment. Each agent is defined by several parameters, such as the resolution of the rendered image that is acquired from the 3D scene, its position and orientation. By means of a few lines of code, an agent can return fully-annotated views of the environment:

```

from sailenv.agent import Agent

agent = Agent(width=256, height=192,
              host="192.168.1.3", port=8085)
agent.register()
agent.change_scene(agent.scenes[2])

while True:
    frame_views = agent.get_frame()
    ...
agent.delete()

```

⁴See <https://unity.com>

The data (`frame_views`) provided by the agent include: the *RGB View* (pixel colors of the rendered scene), *Optical Flow* (motion)⁵, *Semantic Segmentation* (category-level labels), *Instance Segmentation* (instance-level labels), and *Depth View* (depth). Each of these elements contains pixel-wise dense annotations. They are all generated in real-time, and they are then transmitted to the Python client with a fast low-level communication mechanism. This facilitates the use of the SAILenv platform in real-time online learning scenarios.

For the purpose of this work, we extended the SAILenv platform to support dynamic scene generation following the guidelines of Section 4.1. The new Python API we developed also allows the customization of the scene without having to deal with 3D-graphics editing tools or the Unity Editor, creating new objects on-demand.

Scenes and objects. We extended the SAILenv Python API to allow an easy and quick definition of the parameters in Θ , through few lines of code. After having registered the Agent in the environment (as shown in the previous code snippet), a pre-designed scene σ can be chosen using the method `agent.change_scene(scene_name)`. In particular, SAILenv comes with the following scenes, $S = \{\text{object_view (empty space), room01 (bedroom), room02 (living room), room03 (bathroom)}\}$ (see Figure 4.1). Selecting a scene automatically determines the set Ω of available templates. Given a certain template, a new object φ_i can be generated through the method `agent.spawn_object(template_name, position, rotation[, dynamic, limited_to_view])`, specifying its position, rotation and, in the case of a moving object, the properties of the associated trajectory (last two arguments). This method will return an `object_id`. Invoking the creation at frame k will spawn the selected object at the next frame ($k_i = k + 1$) and it will set π_i to the concatenation of the given position and rotation. We postpone the description of the trajectory dynamics (`dynamic` argument) to the next paragraph, while when the Boolean flag `limited_to_view` is set to true, the object will be always kept within the field of view of the agent. The condition for making this choice effective is to create invisible barriers where the object will bounce, located at the borders of the agent camera frustum (that is the region of 3D world seen by the agent), by calling `agent.spawn_collidable_view_frustum()`. The object can then be deleted through the method `agent.despawn_object(object_id)` which is equivalent to setting \hat{k}_i to the identifier of the next frame.

Trajectories. The object dynamics can be defined through simple Python classes. In this work, we propose three different types of trajectories, associated to classes that can be instantiated by calling: `LinearWaypoints(waypoints_list, total_`

⁵A pixel of the Optical Flow View is a vector $(v_x, v_y) \in \mathbb{R}^2$ representing the velocity in px/frame. For visualization purposes (e.g. see the Optical Flow rows of Figures 4.5, 4.6, 4.7), each vector could

time), `CatmullWaypoints(waypoints_list, total_time)` and `UniformMovement_ | RandomBounce(speed, angular_speed, start_direction[, seed])`. Within the notation of Section 2, the chosen class trajectory for the i -th object is what we formalized with κ_i (for example, consider κ_i set to `UniformMovementRandomBounce`), while the associated arguments (`speed`, `angular_speed`, `start_direction`[, `seed`], in the case of the previous example) stand for $\vartheta_i^1, \dots, \vartheta_i^4$. Both `CatmullWaypoints` and `LinearWaypoints` require a list of L waypoints $(w_1, \dots, w_L) \in (\mathbb{R}^6)^L$ and the time (in seconds) that the object takes to loop around all of them, see Figure 4.2 for an example of code (described in the next section). The difference between the two dynamics is that the former does a linear interpolation between two consecutive waypoints, while the latter computes a Catmull-Rom Spline interpolation Maggini et al. (2007) along the whole set of waypoints. Collisions with other scene elements are handled by the Unity physics engine, that takes care of rejoining the trajectory whenever it becomes possible. `UniformMovementRandomBounce` makes an object move inertially until it hits another one or the edges of the agent view. After the collision, the object bounces back in a random direction and also acquires an additional random torque. The `speed` and the `angular_speed` parameters limit the velocity of the object in the scene, the `start_direction` bootstraps its movement and the `seed` may be fixed to replicate the same dynamics (i.e., for reproducibility purposes). Furthermore, the API allows to change the object position and orientation at any given time through the method `agent.move_object(object_id, position, rotation)`. The above presented dynamics are available from the Python API, but with little effort the dynamics can be extended within the SAILenv source code.

Utilities. What we described so far fully defines the scene and the parameters in Θ . In order to simplify the management of the Python code, we added a higher abstraction level based on the Python class `Scenario` and some additional utility classes, such as `Waypoint` and `Object`, that allow to describe the structure of the scene in a compact manner, as we will show in the examples of Section 4.3 (Figure 4.2, 4.3 and 4.4). When using class `Scenario`, the object trajectories can be orchestrated through the `Timings` classes. There are three different available timings. The first one, `AllTogether(wait_time)`, makes every object move at the same time after `wait_time` seconds. The second, `WaitUntilComplete`, supports only waypoint-based dynamics (more, generally, dynamics that are based on loops), and activates them one at a time waiting until each one is complete before starting the next one. Finally, the `DictTimings(_map)` timing takes as input a map that defines for each trajectory how long it should be active before stopping and starting the next one.

be converted in polar coordinates (ρ, ϕ) and the pixel could be assigned the HSV color $(\phi, 1, \rho)$. Therefore, ρ would determine the intensity of the color (the faster, the brighter), while ϕ would determine the color (red: left, green: down, cyan: right, violet: up).

```

scene = "object_view/scene"
waypoints = [
    Waypoint(Vector3(0., 0., 4.), Vector3(0., 0., 0.)),
    ...
    Waypoint(Vector3(-5., 1., 7.), Vector3(90., 90., 180.))
]
dynamic = CatmullWaypoints(waypoints=waypoints, total_time=10.0)
objects = [
    Object("c1", "Cylinder",
          Vector3(0, 0, 2), Vector3(0, 0, 0), dynamic)
]
scenario = Scenario(scene, objects)
agent.load_scenario(scenario)

```

Figure 4.2: A Cylinder moves through the defined waypoints, with a trajectory obtained by Catmull interpolation.

```

scene = "room_02/scene"
dynamic1 = UniformMovementRandomBounce(seed=32,
                                       speed=0.8, start_direction=Vector3(0, 5, 2))
dynamic2 = UniformMovementRandomBounce(...)
dynamic3 = UniformMovementRandomBounce(...)
agent_pos = agent.get_position()
objects = [
    Object("c1", "Chair 01", agent_pos + Vector3(2, 0, 0),
          Vector3(0, 0, 0), dynamic1, frustum_limited=True),
    Object("p1", "Pillow 01", ...),
    Object("d1", "Dish 01", ...)
]
timings = AllTogetherTimings(0.75)
view_limits = Frustum(True, 10.)
scenario = Scenario(scene, objects, timings, view_limits)
agent.load_scenario(scenario)

```

Figure 4.3: Definition of a simple scenario where a Chair, a Pillow and a Dish move pseudo-randomly around a pre-built living room.

Finally, we mention the `Frustum` class to simplify the creation of the previously described invisible boundaries, if needed.

4.3 Examples

The proposed SAILenv-based generator can be downloaded at SAILenv official website <https://sailab.diism.unisi.it/sailenv/>. In the following we show three examples of generations.

```

scene = "room_01/scene"
waypoints = [
    Waypoint(Vector3(0.5, 1.4, 0.5), Vector3(0., 0., 0.)),
    Waypoint(Vector3(0.3, 1., -1.), Vector3(90., 0., 0.)),
    ...
]
agent_pos = Vector3(-1.3, 2., 1.5)
agent.set_position(agent_pos)
agent.set_rotation(Vector3(22., 144., 0))
dynamic = CatmullWaypoints(waypoints=waypoints)
objects = [
    Object("racket", "Tennis Racket 01",
          Vector3(0.5, 1.4, 0.5), Vector3(0., 0., 0.), dynamic)
]
scenario = Scenario(scene, objects)
agent.load_scenario(scenario)

```

Figure 4.4: A Tennis Racket moves along a set of waypoints (Catmull interpolation) inside a pre-built bedroom.

Example 1. In Figure 4.2 the SAILenv basic scene named `object_view` is chosen, that is an empty space with monochrome background. Then, a set of waypoints is defined and the dynamic `CatmullWaypoints` is created using them. A single object is specified, named `c1`, based on template `Cylinder`, at position `Vector3(0, 0, 2)` and with an initial orientation specified by `Vector3(0, 0, 0)` (Euler angles); here `Vector3(_, _, _)` is the description of a three dimensional vector. The `CatmullWaypoints` dynamics will move the `Cylinder` through each waypoint, interpolating the trajectory with a Catmull-Rom spline. Using the notation presented in Section 4.1, we have: $\sigma = \text{object_view}$, $\Omega = \{\dots, \text{Cylinder}, \dots\}$, $\Phi = (c1)$, $(k_1, \hat{k}_1) = (0, \infty)$ and the associated trajectory is specified by $\kappa_1 = \text{CatmullWaypoints}$, $\vartheta_1^1 = \text{waypoints}$ and $\vartheta_1^2 = \text{total_time} = 10$. The generated RGB view and the corresponding optical flow are shown in Figure 4.5 considering four different time instants.

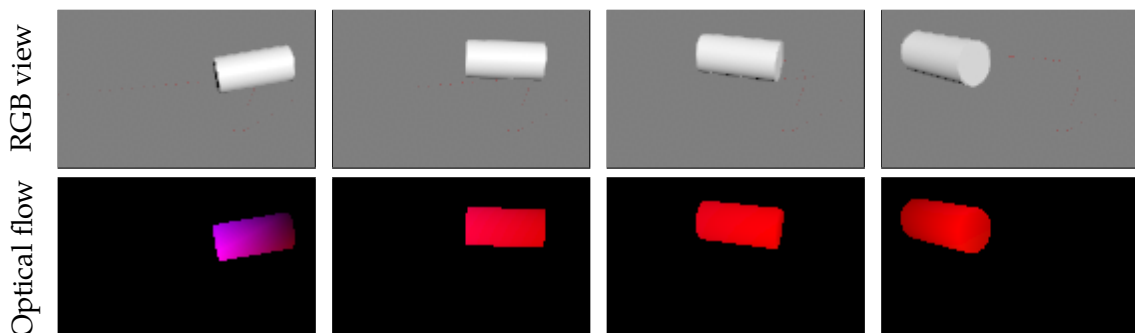


Figure 4.5: Scene described by the script in Figure 4.2. Four frames are shown (from left to right)—RGB view and optical flow.

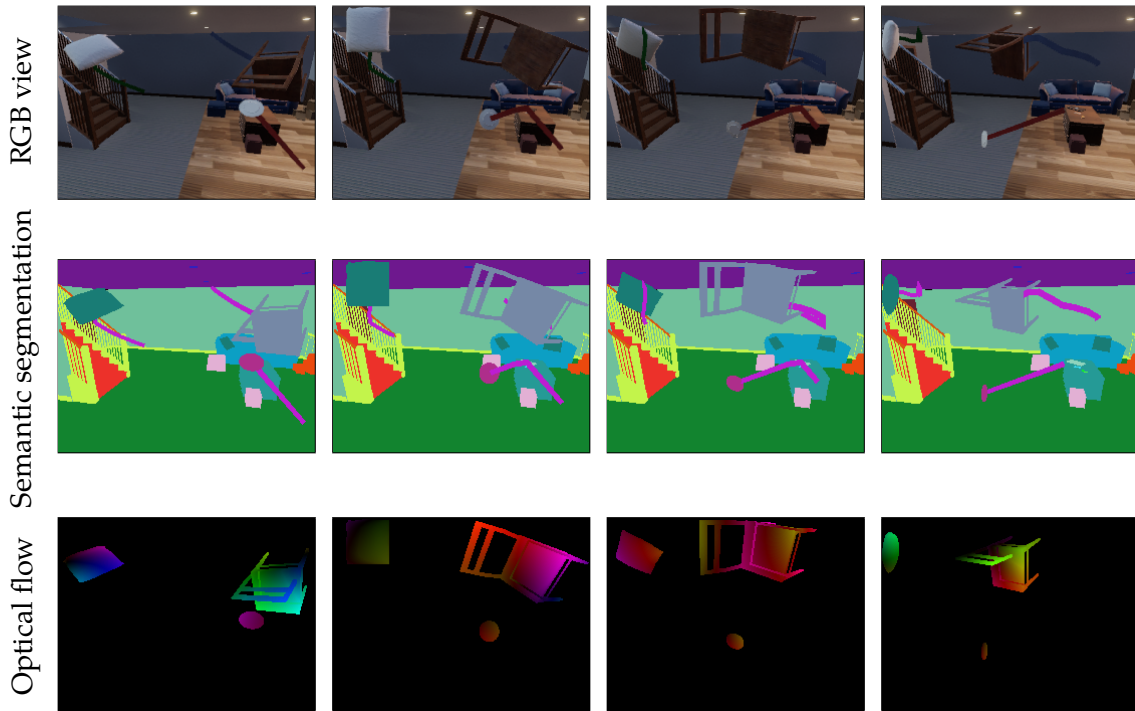


Figure 4.6: Scene described by the script in Figure 4.3 (room02—livingroom scene) considering four different frames (from left to right). For each object, the chosen dynamic is `UniformMovementRandomBounce`. For each frame we display the RGB view, the semantic segmentation and the optical flow. Additionally, we depict in the RGB and semantic segmentation views the local trajectories followed by the moving objects (attached to the moving objects).

Example 2. In Figure 4.3 the selected pre-designed scene is `room_02`, a realistic living room with common furniture. The novel SAILenv API allows us to add new objects that, in this case, are a chair, a pillow and a dish, from the templates `Chair 01`, `Pillow 01` and `Dish 01`. They are initially located in specific points relative to the agent’s position (`agent_pos + Vector3(, ,)`) with a certain orientation (the second `Vector3(, ,)`). For all the objects, the dynamic `UniformMovementRandomBounce` is chosen, also specifying their speed, their initial direction and the seed to ensure the reproducibility of the pseudo-random bounces. Finally, the `AllTogetherTimings` configuration is selected, making every object move at the same time within the view frustum of the agent and also never going beyond 10 meters of distance from the agent itself (`view_limits=Frustum(True, 10.)`). Using the notation of Section 4.1, we have $\sigma = \text{room_02}$, $\Omega = \{\dots, \text{Chair } 01, \dots, \text{Pillow } 01, \dots, \text{Dish } 01, \dots\}$, $\Phi = (c1, p1, d1)$ and $(k_i, \hat{k}_i) = (0, +\infty) \forall i$. Moreover, $\kappa_i = \text{UniformMovementRandomBounce}$ with possible different ϑ_i^m (seed, speed, start_direction) $\forall i$.

For an illustration of the final result, see Figure 4.6 (RGB view, semantic segmentation and optical flow).

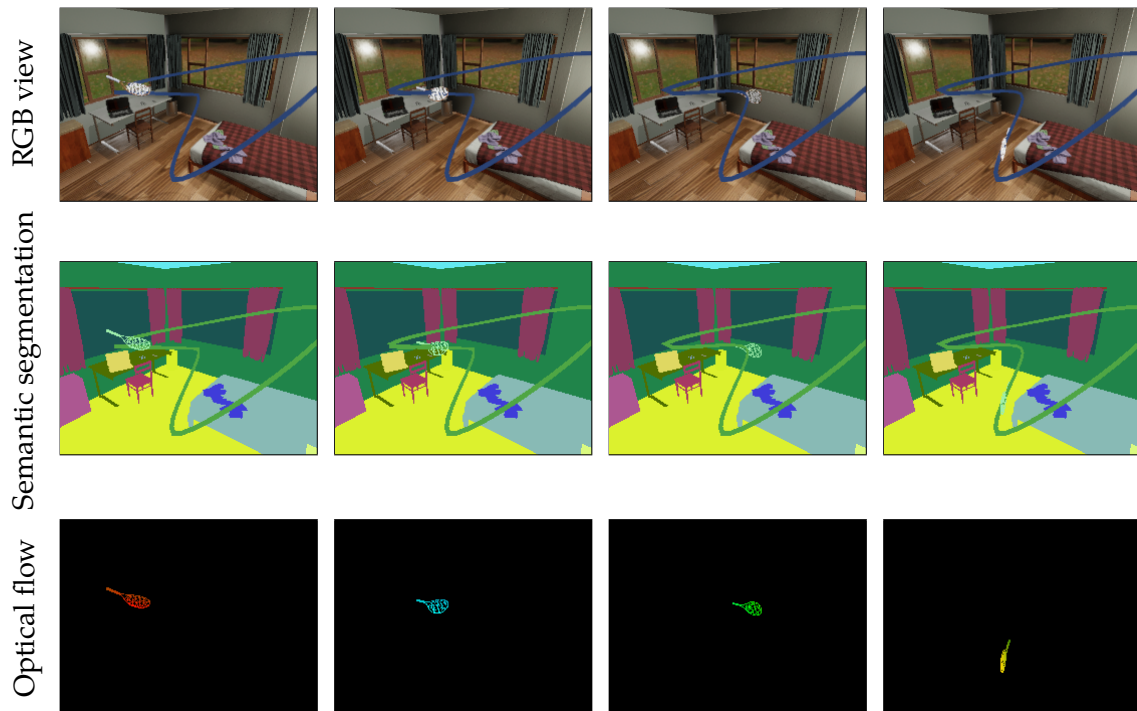


Figure 4.7: Scene described by the script in Figure 4.4 (room01—bedroom scene) considering four different frames (from left to right). The chosen dynamic is `CatmullWaypoints`. For each frame we display the RGB view, the semantic segmentation and the optical flow. Additionally, we depict in the RGB and semantic segmentation views the full trajectory followed by the moving object (attached to the racket).

Example 3. Finally, the code in Figure 4.4 illustrates another realistic scene (bedroom) in which a tennis racket moves according to the `CatmullWaypoints` dynamic. The selected waypoints are defined at the beginning of the script, together with the initial position and orientation of the racket. According to the notation of Section 4.1, we have $\sigma = \text{room_01}$, $\Phi = (\text{racket})$ from the template `Tennis Racket 01` and $(k_i, \hat{k}_i) = (0, +\infty) \forall i$. In this last case, $\kappa_i = \text{CatmullWaypoints}$ and $\theta_1^1 = \text{waypoints}$. The final result is shown in Figure 4.7. Notice that we also used `SAILenv` facilities to change the position and orientation of the agent.

4.4 Discussion

In this chapter we have proposed the idea of generating fully customizable datasets to train and test continual learning agents through the use of 3D-virtual environments. The need of such parametrical framework was motivated through an example scenario where the learning agent takes advantage of the time coherence in a visual stream to learn high-level feature extraction, which is then used to build

a pixel-wise object detector with few supervisions limited to single pixels (Tiezzi et al., 2022b). Describing the generating process of the scenes parametrically allows the user to have full control on the final visual stream the agent perceives and, given a certain learning task, to create scenarios of increasing difficulty. We have reported a concrete realization of these ideas in the SAILenv virtual environment, showing the potential effectiveness of this approach.

The proposed framework has, at this time, some limitations. In particular, it is quite profuse in terms of number of parameters. To create a scene both highly realistic and rich of information, many parameters need to be fixed and tweaked to obtain the required scenario. Future works should focus on simplifying the work needed by the researcher to automatize the definition of the generation parameters. An example in this direction, partially implemented in the source code of SAILenv, is generating the code through graphical manipulation of scenes inside the Unity Editor, which is then translated in code similar to Figure 4.6, Figure 4.5, Figure 4.7. Another possible direction is automatic generation of the scene through some heuristics that take into account both density and diversity of objects in the scene and how they move through the available space. Note that (Gan et al., 2020) already implements the heuristics regarding density and diversity of objects to automatically generate some scenes. Finally, a possible example, which we regard to be possible in the near future, is exploiting a language model to translate natural language requirements of a scene to the code that will generate the scene through the parametrical framework described in the chapter, similar to models such as GitHub Copilot⁶, based on GPT-3 (Brown et al., 2020), can translate software requirements expressed in natural language to functioning code.

⁶Available at <https://github.com/features/copilot>

Chapter 5

Adversarial Attacks in Virtual Environments

As previously stated, in the recent years, the scientific community showed a remarkable and increasing interest towards 3D Virtual Environments, to train and test Machine Learning-based models in realistic virtual worlds. Other than that, these environments could prove very useful as a mean to study the weaknesses of Machine Learning algorithms, or to simulate training settings that allow neural models to achieve greater robustness to adversarial attacks. In particular, for the case of Continual and Lifelong Learning, we see the usefulness of Virtual Environments as habitats of agents that live and learn while interacting with their surroundings. On the other hand, the growing popularity of such tools might also attract those that aim to create adversarial conditions that invalidate benchmarking processes developed through Virtual Environments, or even inject backdoors in Machine Learning systems trained on virtual data. This problem cannot absolutely be overlooked, especially when considering the case of public environments that integrate contributions from a large community of people. As a matter of fact, once an adversarial 3D object has been crafted, it can be plugged into different scenes, spreading out its malicious effect in an exponential manner, affecting every generated data about that same subject, while remaining significantly hard to recognize for a human observer. This is clearly more insidious than when dealing with datasets of static images or videos, where altering some data in an adversarial manner (Biggio and Roli, 2018) will only affect the attacked data, not other samples about the same subject. It is even more subtle in the case of Lifelong Learning agents, which live their life into environments with maliciously crafted 3D objects that can act as backdoors, or more generally as points of weakness, when the agent is finally let into the real world, potentially posing risks to humans and its activities. Of course, the matter can be seen in a more constructive perspective, and it is important to consider that researchers can purposely and explicitly augment 3D Virtual Environments using objects generated in

adversarial contexts, with the aim of training more robust Machine Learning-based models or evaluate their robustness to adversarial conditions.

Most work on Adversarial Machine Learning focus on approaches based on altering static images, and to the best of our knowledge little research has been done in studying how to deal with 3D environments and how a 3D object living in a Virtual Environment should be altered to fool a classifier operating inside the simulation. In (Athalye et al., 2018), rendered views of a 3D object were used for crafting a real 3D object that could consistently make a classifier predict wildly incorrect classes even if a human observers could barely see something wrong on the attacked object.

In this chapter, we study how to craft adversarial 3D objects by altering their textures (i. e. surface colors) using a software tool chain specifically designed to include easily accessible elements. We show that it is possible, and indeed simple, to create adversarial objects with off-the-shelf limited renderers, called *surrogate renderers* in the rest of the document, with the only requirement that they must be able to compute gradients with respect to the parameters of the rendering process. We also show that it is possible, up to a certain extent, to transfer the attacks on much more complex and advanced 3D engines, such as those used in popular Virtual Environments, whose renderers are referred to as *target renderers*, which are considered to not let any kind of gradient flow through the rendering process. Considering the taxonomy presented in section 2.4, this attack can be framed as a White-Box Attack when considering only the surrogate renderer, since the Attacker has complete access and control to the surrogate renderer. When considering transferring the attack to the target renderer, it is instead framed as a Black-Box Attack, since the Attacker does not know anything about the rendering function used. With respect to the neural model, we assume that the Attacker is using a pre-deployed model with no capability of altering its weights, but only to query it and compute the gradient through it. In any case, the attack is framed as a Targeted, Integrity attack, since the aim of the Attacker is to evade correct classification of a given 3D object. We note that, while the term Transferability in many surveys only refer to the property of an Adversarial Example to be effective on a neural model different to that used to craft the example, in this chapter we extend the term to include the capability of an Adversarial 3D Object to be effective even when rendered on a different graphical engine (target renderer) to that used to craft the object (surrogate renderer). To achieve this objective, we propose a saliency-based attack that intersects the two classes of renderers in order to focus the perturbation on those texture elements that we estimate to be most important to the classification when rendered through the target renderer. The saliency is used in conjunction with a maximum perturbation radius to limit the magnitude of the perturbation, to create Adversarial 3D Objects that are as non-suspicious as possible. The need for non-suspicious objects is not strictly fundamental for successful attacks on an object classifier, more so in

a setting where we assume that the Virtual Environment can run without human supervisions, therefore without any human observer that may spot the malicious object, as noted in (Gilmer et al., 2018). If the object is present in a scene during deployment of the Virtual Environment, it may be impractical to spot the object in a scene dense of other entities. However, in our model study, stream generation is not the only scenario in which the Adversarial 3D Objects must avoid detection. Indeed, we assume that the Attacker has not unrestricted access to the development of the Virtual Environment, thus it cannot simply add objects to the Objects Library. Instead, we assume that all contributions, more so for external users contributions, must pass the scrutiny of other maintainers, therefore needing the objects to arouse as less suspicion as possible. Finally, we evaluate the impact of such transferred attacks on popular neural classifiers.

The rest of the chapter is organized as follows. In section 5.1 we briefly introduce the concepts of renderers, both differentiable and non-differentiable, concepts that will be useful in the following sections. In section 5.2 we introduce the theoretical framework on which the proposed Adversarial Attack is founded and finally we describe our proposed saliency-based 3D Adversarial Attack. In section 5.3 we analyze the experimental results yielded by the implementation of the proposed attack. Finally, in section 5.4 we summarize the chapter and discuss possible improvements to the work.

5.1 Renderers: Differentiable and Non-Differentiable

We define *rendering* as the process that takes some formalization of a 3D scene (considering objects, their appearance, the lights, etc) and the parameters of a *camera*, and generates a 2D image which will be the projection of the scene on the visual field of the camera. The rendering is processed by a computer program known as *rendering software*, or *renderer* for short. We can think of rendering as a function r from a 3D scene s and a camera c to a 2D Image $I_{s,c}$.

$$I_{s,c} = r(s, c), \quad (5.1)$$

where any s is composed of 3D objects (meshes), lights, and other elements that are part of the simulation. During rendering, each element in the scene is projected on the camera view plane, whilst taking into account the effects of lights onto the surface of objects relative to the relevant properties of the objects. The nature of the properties available to formalize the scene and the objects are heavily depending on the type of renderer r that we are using. Modern 3D engines support high-end rendering facilities, among which we mentioned Physically Based Rendering (PBR), which indicates a broad range of technologies that simulate the behavior of light impacting and bouncing on the so-called *materials*, that is the set of properties that

describe the surface of a 3D object, mainly its interactions with light, allowing the object to react to light sources in a realistic manner. Each material has specific properties and texture maps that define its roughness, reflectivity, occlusions, and so forth, depending on the engine specifications. For example, the standard shader in Unity3D (Haas, 2014) supports the definition of color and opacity (Albedo Texture Map) and how metallic and smooth the surface of the object should be (Metallic Smoothness Texture Map), together with several other properties (Albedo Map Color, Ambient Occlusion Texture Map, Smoothness Multiplier, Normal Map, etc.). Differently, non-PBR renderers most of the information that is usually contained in a PBR material is held by a single texture called *Diffuse Map*. A Diffuse Map is usually very similar to how the rendered object's PBR material would look if laid on a texture, using the same UV map. The main difference is that the diffuse map cannot react to light, nor can it display any kind of reflection. Diffuse maps are usually hand-made by artists or "baked" within external software. As a matter of fact, generic renderers compute function r of Equation 5.1 by means of non-differentiable operations.

In the last years, the scientific community focused on alternative tools to implement a rendering function. In particular, researchers studied neural models to learn Equation 5.1 from data (Kato et al., 2018; Mildenhall et al., 2020; Rematas and Ferrari, 2020), or, more specifically, they promoted new rendering software that allows the user to compute gradients with respect to several parameters involved in the rendering process (Liu et al., 2019; Nimier-David et al., 2019; Ravi et al., 2020). Among the latter category, we mention PyTorch3D (Ravi et al., 2020), that implement a differentiable rendering API based on the widely diffused machine learning framework PyTorch.¹ Despite being extremely versatile, several differentiable renderers do not support PBR (Liu et al., 2019; Ravi et al., 2020) or other advanced rendering facilities, thus not reaching the level of photorealism that is typical of high-end non-differentiable renderers.

5.2 Adversarial Attacks and Adversarial 3D Objects

The growing diffusion of deep learning methods and applications in real-life scenarios (Grigorescu et al., 2020) poses serious concerns on their robustness. In particular, the vulnerability of their prediction performances to intentionally designed alterations of input data, i.e., adversarial examples (Biggio et al., 2013; Biggio and Roli, 2018; Szegedy et al., 2013), has been proven using several methods, such as Fast Gradient Sign Method (FGSM) (Goodfellow et al., 2014), Projected Gradient Descent (PGD) (Madry et al., 2017) and many others (Akhtar and Mian, 2018).

Let us consider a classification task and a generic annotated pair (x, y) , where

¹See <https://pytorch.org/> and <https://pytorch3d.org/>

$x \in \mathbb{R}^d$ denotes an input pattern and y is the associated supervision. We also consider a neural network classifier $\mathcal{C}(\cdot|\subseteq)$ with parameters $\theta \in \mathbb{R}^p$. Let us indicate with \bar{y} the output yielded by the classifier when processing x , $\bar{y} = \mathcal{C}(x|\theta)$, and the loss function $L(\bar{y}, y, x)$ that measures the mismatch between the prediction and the ground truth. A common learning procedure aims at identifying the model parameters θ which minimize the empirical risk function $\mathbb{E}_{(x_k, y_k) \sim \mathcal{X}} [L(\hat{y}_k, y_k | x_k)]$. Neural classifiers have been proved to be vulnerable to the injection of adversarial perturbations in the input space, resulting in the misclassification of the pattern at-hand. In particular, an adversarial input $x + \delta$ causes $\mathcal{C}(\cdot|\theta)$ to make wrong predictions, i.e., $\hat{y} = \mathcal{C}(x + \delta|\theta)$ with $\hat{y} \neq y$. In order to inject a perturbation δ that can be considered imperceptible to humans, a set of admissible perturbations \mathcal{P} is defined, which in the context of image classification is guided by visual perceptibility. A common choice limits the perturbation to fall upon a ℓ_2 -ball or in a ℓ_∞ -ball, and henceforth we will consider the latter. In the most simple case, the goal of the attacker is to find δ as solution of the following optimization problem,

$$\max_{\delta \in \mathcal{P}} L(\bar{y}, y, x + \delta). \quad (5.2)$$

In the specific case of FGSM, adversarial examples are computed as

$$\hat{x}_k = x_k + \varepsilon \cdot \text{sgn}(\nabla_{\delta_k}) \quad (5.3)$$

In the specific case of PGD, the problem in Equation 5.2 is solved by an iterative scheme, eventually including random restarts,

$$x^{k+1} = \Pi_{\mathcal{P}}(x^{tk} + \alpha \cdot \text{sign}((\nabla_x L)(\bar{y}, y, x^k))) \quad (5.4)$$

being $x^0 = x$,² t the iteration index, $\alpha > 0$ the step length and $\Pi_{\mathcal{P}}$ projects its argument onto an ℓ_∞ -ball with radius ε centered on the original example.

In the case of 3D data, prior work (Lu et al., 2017; Luo et al., 2015) has shown that carefully-designed 2D adversarial examples fail to fool classifiers in the physical 3D world under several image transformations, such as changes in viewpoint, angle or other conditions (camera noise or light variation). In order to generalize attacks to such contexts, Athalye et al. (Athalye et al., 2018) proposed adversarial examples that are robust over a certain distribution of transformations. The aforementioned insensitivity of the attack to changes of viewing conditions is attained optimizing the problem in Equation 5.2 with respect to the expectation over a distribution \mathcal{T} of transformation functions t , denoted as *Expectation over Transformation* (EOT), applied on the adversarial pattern, $t(\hat{x}_k)$. The transformed adversarial pattern is the one which is actually perceived by the classifier. The optimization problem is transformed into:

$$\max_{\delta \in \mathcal{P}_{\varepsilon^t}} \mathbb{E}_{t \sim \mathcal{T}} [L(\bar{y}, y, t(x + \delta))] \quad (5.5)$$

²Or $x^0 = x + \delta^0$, with δ^0 that is randomly generated.

being \mathcal{T} a distribution of transformations and $t(\cdot)$ is a transformation sampled from \mathcal{T} , while $t(x + \delta)$ is the input of the classifier. Moreover, $\mathcal{P}_{\varepsilon'}$ is the set of perturbations, where we define $\mathbb{E}_{\mathcal{P}}$, given a distance function $d(\cdot, \cdot)$, as

$$\mathbb{E}_{t \sim \mathcal{T}}[d(t(x + \delta), t(x))] < \varepsilon' \quad (5.6)$$

where $d(\cdot, \cdot)$ is a distance function and $\varepsilon' > 0$. This approach basically introduces expectations both in the objective function and in the perturbation-related constraint. Equation 5.6 assures that the optimization process minimizes the expected distance which is perceived by the classifier. What is important to consider is that \mathcal{T} is about a wide variety of transformations, including special operations that consist in using $x + \delta$ as a texture of a 3D object and rendering it to a 2D image. The authors of (Athalye et al., 2018) use this intuition to physically create real-world 3D objects that are adversarial over different visual poses. Of course, this requires a renderer (section 5.1) that is differentiable, and (Athalye et al., 2018) is based on specific ad-hoc operations that cannot be easily implemented in a general setting.

Note that the considered family of transformation \mathcal{T} could express heterogeneous distortions such as random affine transformations as long as noise addition or operations such as rendering $t(x)$ of a texture x . In particular this last consideration is exploited in (Athalye et al., 2018) to craft textures that are adversarial over distributions of visual poses of the corresponding 3D real-world object. In order to do so, EOT requires a the differentiability of the renderer, which is obtained by some shortcuts such as studying an approximation of the derivatives for a particular renderer or using neural renderers. Leveraging this method, the authors are capable to craft a texture the rendering of which is adversarial from any viewpoint.

Other recent works prove that the texture space is not the unique element which can be attacked in the 3D setting. In (Zeng et al., 2019), authors create adversarial attacks using both differentiable and non-differentiable renderers. In the former case, they perturb multiple physical parameters such as the material, the illumination or the normal map, in Visual Question Answering (VQA) and 3D shape classification. Notice that the camera parameters are assumed to be fixed, hence producing a single projection on the 2D-perceptual space which becomes adversarial. This could result in implausible shapes. Liu et al. (Liu et al., 2018) proposed perturbations that are focused on lighting. Their work, rather than being based on the concept of norm-balls around pixels, leverages a parametric norm-ball based on the parameters that guide the image rendering. This is possible thanks to the introduction of a physically-based differentiable renderer capable to backpropagate gradients to the parametric space, focused on the light aspect. MeshAdv (Xiao et al., 2019) alters the object meshes, using a neural renderer applied on models with constant reflectance, leading to very simple textures. The authors investigate the robustness under various viewpoints and the transferability to black-box renderers under controlled rendering parameters. A recent work by Yao et al. (Yao et al., 2020) leverages multi-view

attacks inspired by EOT, in order to devise 3D adversarial objects, perturbing the texture space and investigating the attack quality using multiple classifiers. Finally, Liu et al. (Liu et al., 2020) considers the case of embodied agents performing navigation and question answering. To better attack the task at hand, the perturbations are focused on the salient stimuli characterizing the temporal trajectory followed by the embodied agent to complete its task.

We want to focus on the problem of generating adversarial 3D objects in the context of the 3D Virtual Environments. The existing experiences in crafting 3D adversarial objects have shown that it is indeed possible to create attacks that fool the classifier of a rendered scene. However, existing works are strongly based on ad-hoc solutions, sometimes using specifically created renderers, limiting the attacks to a single view or considering very simple textures. They usually assume that the attacker has access to low-level properties of the renderer, such as the mapping of the view-space coordinates to the texture-space coordinates, or that renderers can be modified to expose additional information (Athalye et al., 2018). Unfortunately, all these assumptions does not make their findings easily adaptable to more general cases. Another remarkable limit is that rendering engines (see section 5.1) are pieces of software that requires advanced skills not only in programming, but also in computer graphics, in order to be modified to accommodate attack procedures, or they might not be open source.

We focus on a more generic perspective, that is based on a realistic setting in which the attacker has the goal of creating adversarial objects for a certain *target renderer* on which he has limited control. We assume attacker to have some skills in Adversarial Machine Learning but not necessarily an advanced knowledge of computer graphics. We explore the idea of synthesizing 3D adversarial objects using off-the-shelf popular software packages, well assembled into a specifically designed tool chain, with the goal of being able to craft malicious examples that can then be transferred to the target renderer of the considered 3D Virtual Environment. We report the structure of the proposed tool chain in Figure 5.1.

Our computational pipeline takes into account two different renderers (Figure 5.1, white boxes). One of them is the already introduced target renderer, while the other one is what we refer to as *surrogate renderer*. The latter is a differentiable renderer on which the attacker has complete control, a reasonable assumption considering that many open-source differentiable renderers have been recently made available to the research community. As discussed in section 5.1, it is likely that differentiable renderers will not perfectly match the quality of the target renderer, so that we focus on the specific case in which there is an evident difference between the outcome of the two renderers. Of course, we assume that the 3D Virtual Environment allows users to introduce and render custom objects. However, care must be taken in adapting the object data format between the two renderers, since there might be a

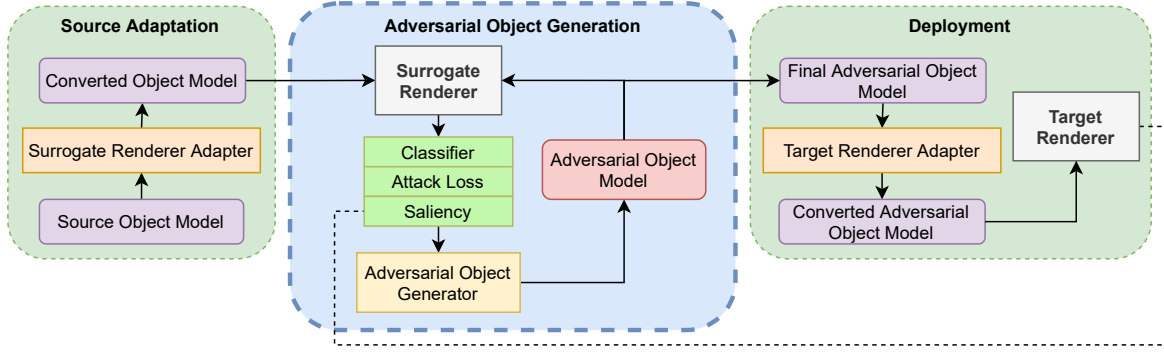


Figure 5.1: Structure of the proposed adversarial object generation procedure. Saliency is computed exploiting the target renderer, as highlighted by the dotted line.

misalignment between the type of models expected by the 3D Virtual Environment and by the surrogate engine, requiring specific adaptations (Figure 5.1, leftmost and rightmost blocks).

Let us consider a certain object o of class y , a scene s and a camera c . The notation o indicates the object and all its properties (mesh, textures, etc.), and, for the sake of simplicity, we indicate with $o + \delta$ an alteration of the object obtained by perturbing its properties by an offset δ . We consider the image (view) $I_{s,c,o}$ that we get when plugging o into scene s , and rendering the whole 3D data when observed from camera c . We overload the notation of r in Equation 5.1 to introduce the dependence on o ,

$$I_{s,c,o} = r(s, c, o). \quad (5.7)$$

A neural network classifier \mathcal{C} (Figure 5.1, top green box) processes $I_{s,c,o}$. The classifier prediction $\bar{y} = \mathcal{C}(I_{s,c,o} | \theta)$ is evaluated into a loss function (Figure 5.1, mid green box) that drives the generation of the adversarial object, inspired by the EOT of Equation 5.5, even if fully focused on transformations in the 3D world. In particular,

$$\max_{\delta \in \mathcal{P}} \mathbb{E}_{(s,c) \sim \mathcal{S}} [L(\bar{y}, y, I_{s,c,o+\delta})] \quad (5.8)$$

where \mathcal{S} includes different camera positions and orientations, different lighting conditions and, in the most generic cases, different backgrounds. Equation 5.8 is paired with a norm-based constraint that ensures $\|I_{s,c,o} - I_{s,c,o+\delta}\|_{\infty} < \epsilon$, for all s, c . This view-based constraint acts as an indirect measure to ensure that o is not changing in a too evident way. Whenever we will need to distinguish between views generated by the surrogate and target renderers, we will use the notation $I_{s,c}^S$ and $I_{s,c}^T$, respectively. The loss L is defined as a cross entropy loss which ignores incorrect classifications, therefore only attacking views where the classifier correctly classi-

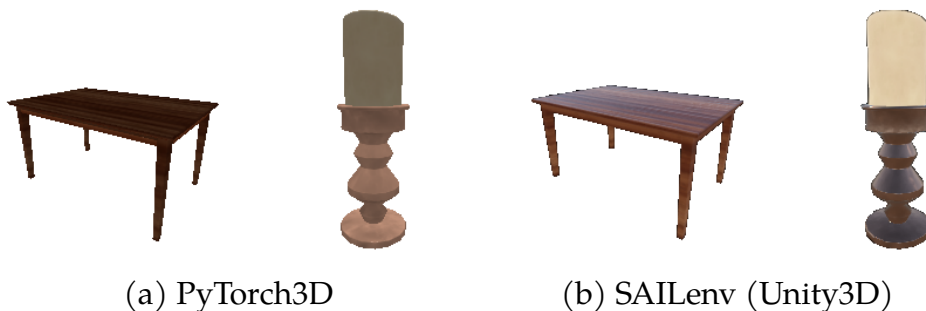


Figure 5.2: Rendering capabilities of the surrogate (a) and target (b) renderers.

fies the object. The loss is then averaged over all views of the object, obtaining an estimation of the EOT in Equation 5.5.

5.3 Experimental study

We instantiated the strategy of Figure 5.1 into a specific case study, that will also drive our experiments. Our choices are completely driven by simplicity, selecting tools that are recent, freely available, and that do not require advanced skills in computer graphics. In particular, we considered SAILenv as the Virtual Environment of choice for the case study. As mentioned in chapter 3, SAILenv exploits Unity3D, which will be our target renderer and satisfies the assumption of allowing the attacker to render custom objects while having limited low-level control to the renderer, since it is based on proprietary code and cannot be “easily” modified. As surrogate renderer we focused on the recent PyTorch3D (Ravi et al., 2020) (section 5.1), that is completely based on Autograd and thus trivial to integrate with a PyTorch-based classifier for gradient computation. The two renderers, beside having different possibility of access to their inner working, have some remarkable differences. SAILenv uses PBR while PyTorch3D is based on diffuse-based rendering, which takes into account only the surface color and a much simplified light model. Both are discussed in section 5.1. We approximate the diffuse maps rendered by Pytorch3D with the Albedo Texture Maps used within Unity3D. This approximation holds the best for neutral illumination settings and for low reflective materials. See Figure 5.2 for a comparison of the rendering capabilities of the two renderers. PyTorch3D allows gradients estimation of several parameters of the object and of the scene (surface color, object geometry, lighting, etc.). For the scope of this paper, we will focus only on the surface color texture. This is a very challenging setting due to the aforementioned limited rendering facilities of PyTorch3D, making this case study a very good representative of the previously described attack scenario. In facts, we expect this to lower the transferability of the attack by a certain degree,

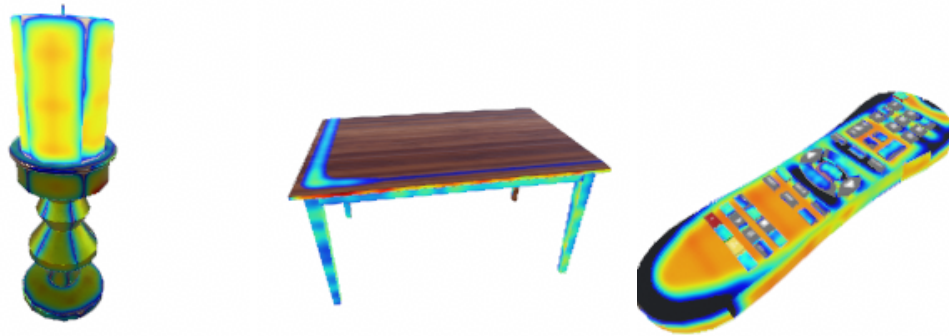


Figure 5.3: Multi-view saliency maps (target renderer), projected into the surface of the object (projection computed by the surrogate engine) – red=high; blue=low.

but that even using a much simpler Surrogate Renderer with respect to the Target Renderer, we can still obtain effective attacks that will work in more realistic settings.

We qualitatively show in Figure 5.3 how the saliency maps, computed using Unity3D over multiple views, are projected back onto the texture space, accumulating their contributes on the texels, then rendering the 3D objects. While computing this projection in the target renderer is not straightforward, this can be easily done following the texturing routine of PyTorch3D, and that is how we created the figure. We can appreciate how the larger saliency areas only cover a subportion of the texels. Notice that the data adapters or Figure 5.1 (i.e. Surrogate Renderer Adapter and Target Renderer Adapter) play a crucial role in our case study, since Unity3D stores objects in a different format (FBX) than the one used by PyTorch3D (OBJ). Therefore, the tool-chain we developed takes account of performing the correct conversions when transferring the attacks from one renderer to the other. We implemented a source object converter by means of a Blender-based³ script, created from scratch. The final adversarial object is then converted back to the Unity3D format through a plugin that is internal to Unity3D, and finally rendered in SAILenv.

In order to evaluate the impact of our strategies in different networks, we selected two popular and powerful deep neural image classifiers trained on ImageNet, that are INCEPTIONV3 and MOBILENETV2.⁴ The former is a state-of-the art image classifier and was chosen because it has the best accuracy on ImageNet, the latter is a smaller model, still very accurate but with faster classification and sensible to use on real-time processing of a video stream from a Virtual Environment. We considered 10 different objects from the SAILenv library, associated to classes that are supported by the classifiers. The objects are: Candle, Teapot, Floor Lamp, Paper

³<https://www.blender.org/>

⁴https://pytorch.org/hub/pytorch_vision_mobilenet_v2/
https://pytorch.org/hub/pytorch_vision_inception_v3/

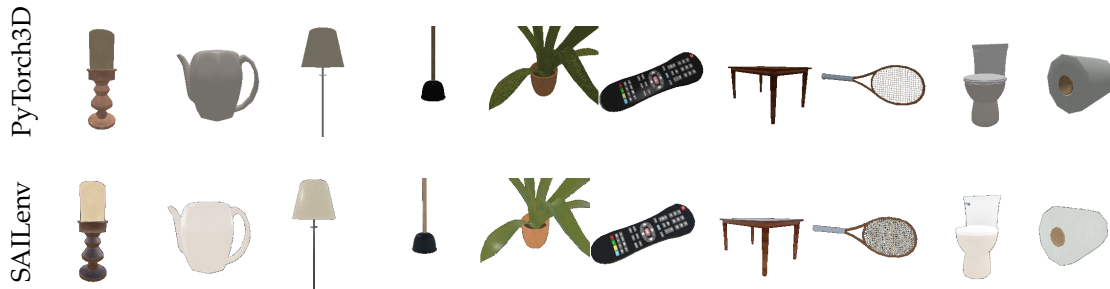


Figure 5.4: Objects considered in our case study, rendered using PyTorch3D (top) and in SAILenv/Unity3D (bottom). In order: *candle, ewer, lamp, plunger, potted plant, remote control, table, racket, toilet, toilet tissue*

Tissue, Plunger, Pot, Remote Controller, Living Room Table, Tennis Racket, Toilet. The objects are shown in Figure 5.4, comparing their appearance in the surrogate and target renderers.

As adversarial object generation method, we implemented the PGD attack described in section 5.2, using the cross entropy loss, parameterized by the parameters ϵ , the maximum L_∞ norm of δ and α , the learning rate of the PGD attack, and τ_S , the saliency threshold. The saliency maps are computed on the images rendered by SAILenv. After applying the threshold, they are used as a mask to stop the gradients flowing from PyTorch3D renderings. An example of the Saliency Maps before applying the threshold can be seen in Figure 5.3. The attack follows these steps:

1. Saliency maps are optionally computed on all the views of the object rendered by the Target Renderer
2. Accuracy pre-attack is computed, classifying the renderings of the object from all viewpoints, both for the Surrogate Renderer and the Target Renderer.
3. A PGD Attack is performed, by rendering the object from all viewpoints on a plain constant background and trying to increase the classification loss by changing the texture with a perturbation δ . The gradient is allowed to flow only from those pixel whose saliency is above a certain threshold.
4. Accuracy post-attack is calculated similarly to step 2, obtaining the *accuracy drop* for both Surrogate Renderer and Target Renderer.

We performed several experiments to evaluate the proposed attack strategy in this case study⁵. Each object is rendered from 60 different views, keeping the camera at a fixed distance which was manually chosen to obtain an iconic image of the object,

⁵Our implementation of what we propose and study in this paper can be found at <https://github.com/sailab-code/SAIFooler>. The 3D models used in the experiments can be found at <http://sailab.diism.unisi.it/sailenv/>.

i. e. so that the object covers most of the picture. The camera turns around the object, from 0° to 360° and also changes its elevation. The range on which the elevation is changed is manually chosen for each object in order to avoid unnatural viewing orientations that would lower the classification accuracy even without any attacks. In facts, we notice that there are several viewpoints from which even INCEPTIONV3 is unable to recognize some of the objects, phenomenon mostly due to the fact that the training dataset contains some objects in a handful of common viewpoints. For an example, see Figure 5.5. Similar results have been recognized and demonstrated in (Alcorn et al., 2019), where they argue that neural networks are easily confused by object rotations and translations. Therefore, we avoid views in which the classifier is already known to have low performances, focusing exclusively on those on which it has good prediction accuracy. We considered a directional light, similar to the way sunlight shines on objects, coming from the front and at an elevation of 75° . The background scene of each object is composed of a uniform color, that was evaluated as being white or black, selecting the one that maximized the recognition accuracy.

We explored attacks that progressively yield larger alterations in the original textures, considering $\varepsilon \in \{0.05, 0.1, 0.5\}$, comparing cases in which we do not use saliency maps or when the maps are binarized with different thresholds of tolerance, i.e., $\tau_s \in \{0.05, 0.2\}$, and we set α to 0.01. It is important to remark that we are considering the ℓ_∞ norm to bound the perturbations, so that, given the same ε , we can have very different number of altered texels. Altering less texels is expected to reduce the probability of letting humans recognize the adversarial object, and that is the goal of the proposed saliency-map-based procedure. We used two metrics to evaluate the quality of the adversarial objects. The first one is the *accuracy drop* A_{drop} , that is the ratio of the variation of average accuracy (before and after the attack, referred to as A_{before} and A_{after} , respectively) to the initial accuracy, while the second one is the percentage of texels $N_\%$ that are altered by the attack procedure. Formally,

$$A_{drop} = \frac{A_{before} - A_{after}}{A_{before}}$$

$$N_\% = \frac{\|\text{tex}_{before} - \text{tex}_{after}\|_1}{|\text{tex}_{before}|},$$

being tex . the texture tensor composed of $|\text{tex}_{before}|$ elements and $\|\cdot\|_1$ the ℓ_1 norm. We computed both the metrics within the PyTorch3D renderer and the SAILenv (Unity3D) renderer. In the former case, we are basically exploring a white-box scenario, where the system we attack is the one on which we evaluate the result. In the latter case, we consider the impact of the adversarial object once it is transferred to a target environment, in a very challenging black-box setting, due to the previously described differences between the two renderers.

In Table 5.1 we report the main results of our experiments, showing A_{drop} for

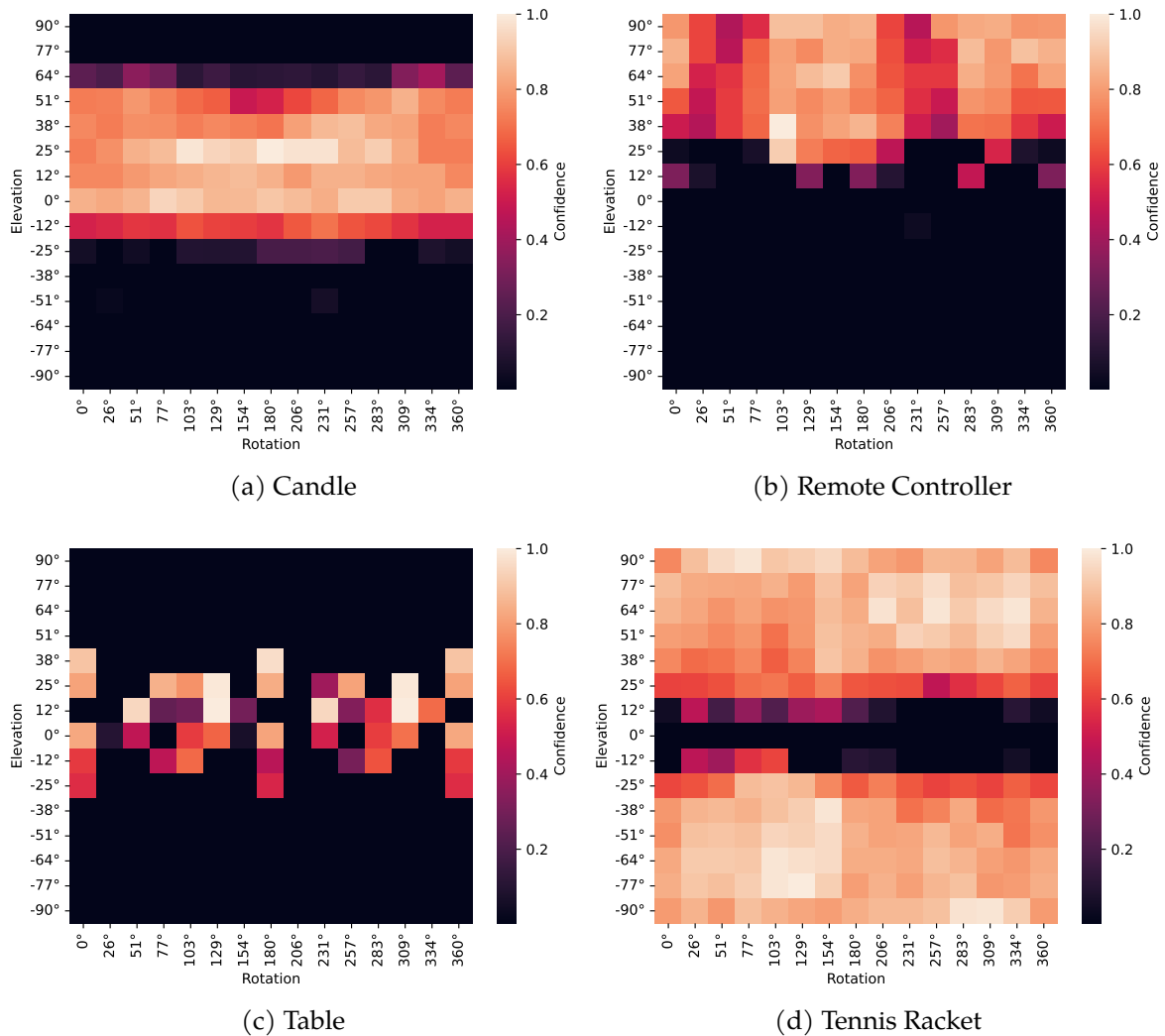











Figure 5.5: The heatmaps show the classification score given on the correct class by INCEPTIONV3 for different values of elevation of the camera and the rotation of the object. As can be seen, objects with a medium degree of rotational symmetry, e. g. Candle (Figure 5.5a) and Tennis Racket (Figure 5.5d), are recognized with high classification scores are high across all rotations of the object. Objects that are commonly seen from above, such as Candle (Figure 5.5a) and Remote Controller (Figure 5.5b) suffer from significantly low classification scores from viewpoints that stay underneath the objects. Finally, an object like Table (Figure 5.5c) is usually captured in the training datasets slightly from above, making it impossible for the resulting network to recognize the Table from higher elevations, and harder to recognize them from unusual orientations of the table (in Figure 5.5c we see that the highest classification scores are slightly from above and at rotations of 0° , 129° , 180° , 309° and 360° , which loosely correspond to viewpoints that are parallel to one of the sides of the table or with approximately 120° or 45° rotation from the camera axis).

Table 5.1: Accuracy drop in each considered object and average result.

		ϵ										Avg	
PyTorch3D	MOBILENETV2	0.05	1.00	n.a.	0.86	1.00	1.00	1.00	0.98	0.95	0.92	1.00	0.97
		0.10	1.00	n.a.	0.86	1.00	1.00	1.00	0.98	1.00	1.00	1.00	0.98
		0.50	1.00	n.a.	0.86	1.00	1.00	1.00	1.00	0.95	1.00	1.00	0.98
	INCEPTIONV3	0.05	1.00	1.00	0.98	1.00	1.00	0.97	0.95	1.00	0.05	1.00	0.89
		0.10	1.00	1.00	0.95	1.00	0.98	0.97	1.00	0.97	0.45	1.00	0.93
		0.50	1.00	1.00	1.00	1.00	1.00	0.94	0.97	1.00	0.97	1.00	0.99
SAILenv	MOBILENETV2	0.05	0.76	n.a.	0.62	1.00	0.72	0.60	0.21	0.00	0.81	n.a.	0.59
		0.10	0.72	n.a.	0.62	1.00	0.74	0.65	0.43	0.00	0.90	n.a.	0.63
		0.50	0.76	n.a.	0.62	1.00	0.74	0.68	0.41	0.00	0.94	n.a.	0.64
	INCEPTIONV3	0.05	0.37	0.87	-0.12	0.65	0.07	0.00	0.10	0.16	0.00	1.00	0.31
		0.10	0.41	0.77	-0.15	0.65	0.08	0.21	0.34	0.47	0.00	1.00	0.38
		0.50	0.39	0.73	-0.08	0.65	0.12	0.09	0.41	0.42	0.07	1.00	0.38

the considered objects and the average result (last column; we indicate with *n.a.* those objects that were not correctly recognized by MOBILENETV2 in their original state). In the case of the surrogate renderer, it is evident that even lower values of ϵ are enough to usually achieve near 100% drop of accuracy, with the exception of the Tennis Racket, for which a higher ϵ is needed. As we can expect, the attack on MobileNet is generally more effective with respect to Inception, as the former is a less robust model compared to the latter.

When the attack is transferred to SAILenv, we can observe that the classifiers are still fooled in a non-negligible manner. Of course, the extent to which the attack has effect is reduced, as expected, but it is surprisingly to see that even if the difference between the two renderers in our case study is significant, the attack can impact the outcome of the classification in the target 3D Virtual Environment. With the exception of Lamp Floor, Pot, Tennis Racket in the case of INCEPTIONV3, and Teapot, Living Room Table for MOBILENETV2, where the attack yields no evident accuracy drops (in one case also a negative drop, meaning that it is slightly improving the classification), the other adversarial objects reduce the accuracy of the classifiers, with a pretty strong effect in the case of MOBILENETV2. It is also interesting to note that using saliency maps usually has a less significant reduction on the A_{drop} if compared to the high reduction inflicted on the A_{drop} on PyTorch3D. This, associated with the showing of Figure 5.6, which will be discussed later, indeed show that saliency can be used as a good trade off between how effective and detectable the attack is.

In Fig. 5.6, we report the 2D plot of A_{drop} against $N_{\%}$ (all objects), taking into account different values of ϵ and saliency thresholds τ_S . When using PyTorch3D, several points are clustered on the right side of the plot, associated to a large A_{drop} . In the case of SAILenv and INCEPTIONV3 as a classifier, the majority of points are between 0.1 and 0.5, with some attacks reaching very large drops. We also note that

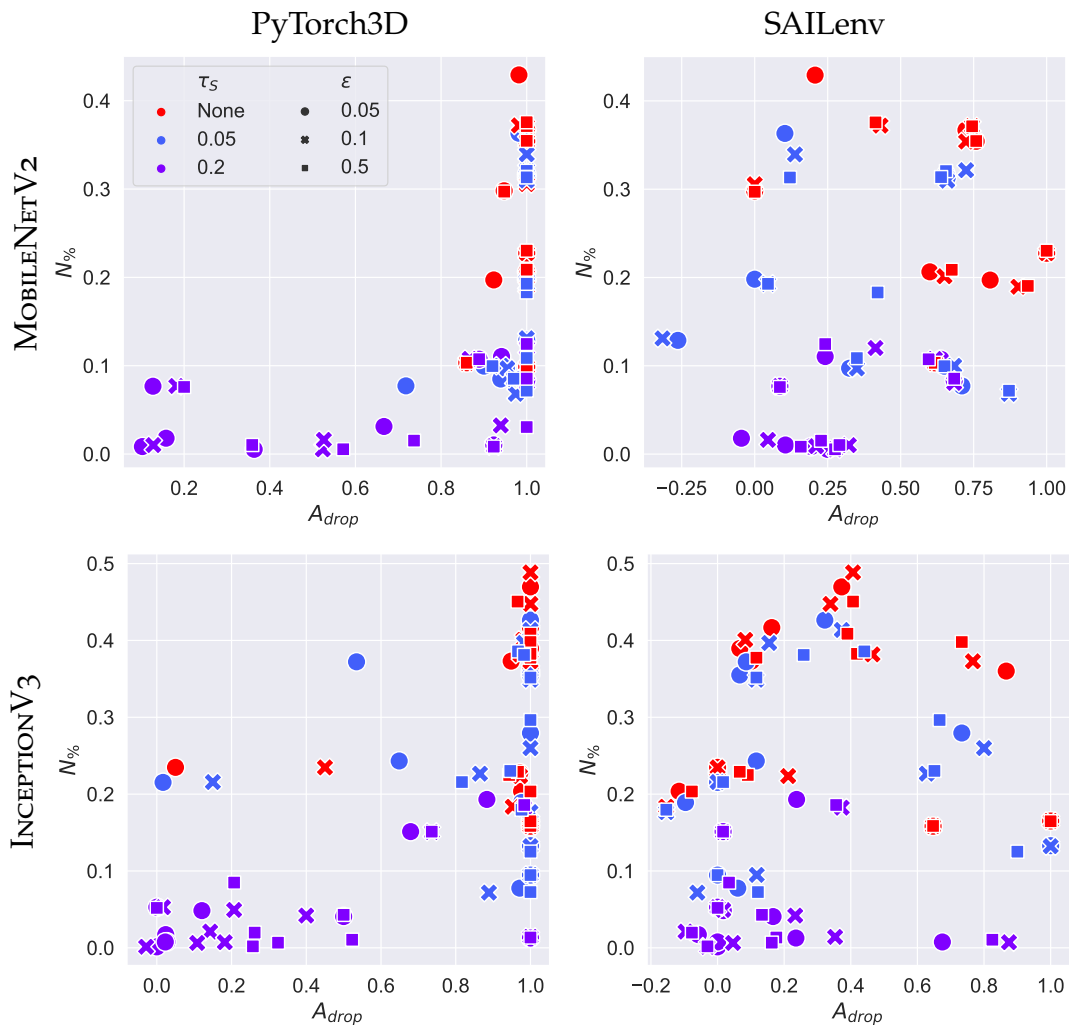


Figure 5.6: Accuracy drop versus percentage of altered texels. Points are about adversarial objects (colors indicate different τ_S ; markers are about different ϵ).

a low number of attacks have an improvement in predictions, shown by a negative A_{drop} . In the case of SAILenv and MOBILENETV2, more attacks have A_{drop} approaching 1.0. As already discussed, even small ϵ might end up in altering a significant amount of texels. However, the plots show that using saliency is a good solution to identify a trade off between A_{drop} and $N_{\%}$. In particular, the attacks in which no saliency information is used are usually located in the upper-right quadrant of the plot – high impact but they heavily alter the textures. Attacks with the largest saliency threshold τ_S are instead usually located in the lower-left quadrant – low impact but they are also more hardly noticeable by humans, altering less pixels. When using a lower τ_S we get results distributed in the central part of the plot – good impact on the classifier, altering a relatively small number of texels.

We qualitatively evaluated the renderings of the adversarial objects, reporting

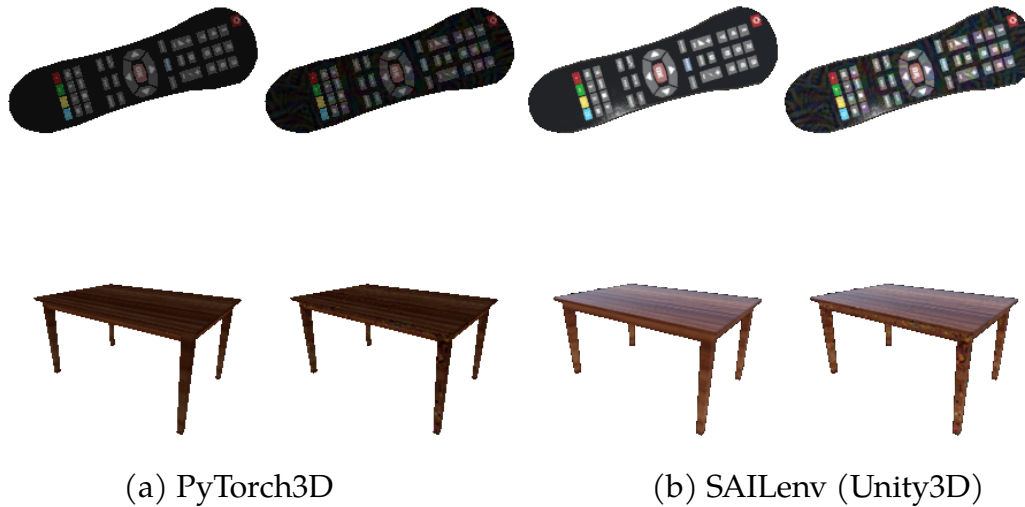


Figure 5.7: For the surrogate (a) and target (b) renderers, we report two objects (one per line), before (left) and after (right) the attack.

in Figure 5.7 two examples that fool the `INCEPTIONV3` classifier in both the renderers. The adversarial Remote was created without using any saliency and ϵ set to 0.1, while the adversarial Table was the outcome of using saliency (τ_S equal to 0.05) and a significantly larger ϵ (0.5). By visually inspecting and comparing the original and the altered objects, it is evident that the two objects have minimal differences that can easily pass unnoticed by a human observer and surely do not make the appearance of the object unrecognizable. Nonetheless, we can still observe the huge gap between what the surrogate and the target renderer produce, remarking that even with these differences, the attack performed on the surrogate can effectively transfer producing a non-negligible drop in performances. Indeed, the classifiers are heavily influenced by the perturbation in the textures, as exemplified by the histograms in Figure 5.8, where we consider all the 60 views of such objects, reporting how the predictions of `INCEPTIONV3` are distributed. It is evident that before the attack, most of the predictions are correctly distributed on the ground truth class, while after the attack they are spread over multiple incorrect classes.

5.4 Discussion

We presented a novel study on the transferability of adversarial 3D objects, created using an off-the-shelf differentiable renderer and then moved to a powerful 3D engine that is at the basis of several recent 3D Virtual Environments. Our analysis showed that it is indeed possible to setup a tool chain based on simple elements that do not require advanced skills in computer graphics, and use it to craft malicious 3D

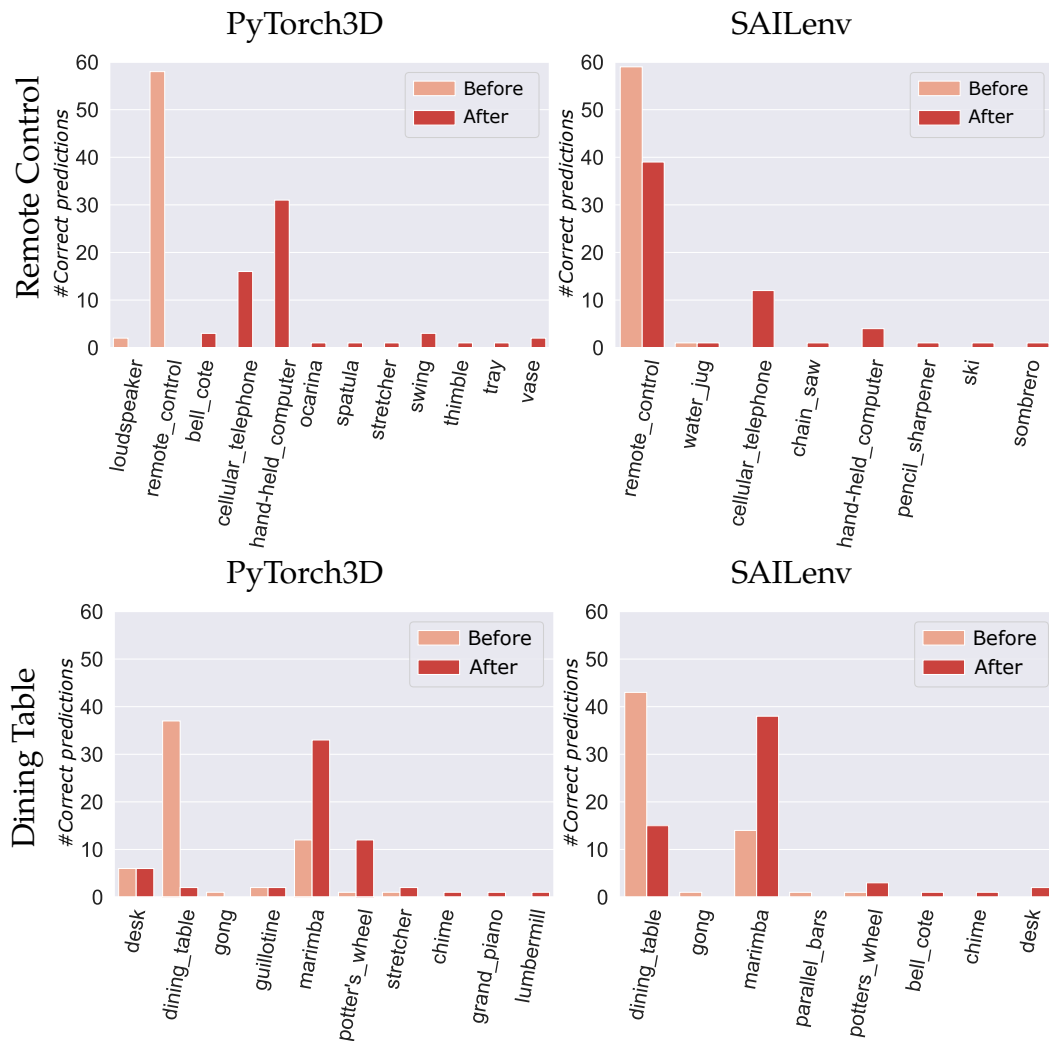


Figure 5.8: Number of correct predictions made by INCEPTIONV3 out of 60 different views of the objects of Fig. 5.7, before and after having attacked them.

objects. Experiments on texture-oriented manipulations showed that attacks can be transferred to fool popular neural classifiers, also considering an estimated saliency of the texels. This demonstrates that Virtual Environments are not exempt from the dangers of Adversarial Attacks, and proper policies and care should be given when accepting public contributions to the object library of a Virtual Environment. There is certainly room for future work in improving the effectiveness of the attacks (e.g., considering other parameters of the renderer – mesh and others). Another possibility, more focused on the target renderer, is to apply black-box optimization algorithms on the Adversarial 3D Object produced by PyTorch, fine-tuning the attack to be even more effective on the target renderer. However, our results are expected to point the attention of the scientific community towards this double-sided aspect: on one hand, it could be an issue for community-open 3D Virtual Environments,

and, on the other hand, it is an opportunity to create even more powerful testing environments, purposely populated with adversarial examples.

Chapter 6

Parallel Computations in Learning from a Video Stream

In the last few years, the Machine Learning community strongly increased its attention towards those learning problems that are framed as *continual* or *lifelong* (DeLange et al., 2021). Even if there exists a large number of recent approaches in such a research direction, this learning setting is still extremely challenging. Real-world applications that are well-suited for continual learning are those that have access to a continuous stream of data, where an artificial agent is not only expected to use the data to make predictions, but also to improve itself and to adapt to changes in the environment, i.e. videos, stream of texts, data from a network of sensors, etc. (Betti et al., 2020c; Maggini et al., 2020). In the case of neural networks, the most extreme and challenging context is one in which a simple online update of model parameters is applied at each time instant, given the information from the last received sample (Betti et al., 2020b).

Despite the importance of having access to powerful computational resources for Continual Learning-based applications, current algorithmic solutions have not been paired with the development of software libraries designed to speed-up learning and inference. The increasing popularity of Lifelong Learning has not been paired with the development of software libraries designed to speed-up learning and inference, in settings in which the data is streamed over time and the model is expected to react to each received sample. As a matter of fact, storing and processing portions of the streamed data in a batch-like fashion is the most common way to approach the problem, reusing classic non-continual learning tools. However, the artificial nature of this approach is striking. Indeed, humans perceive the environment around them as a continuous multimodal stream of data, experienced and processed in real-time, online, without storing huge collections of data and re-experiencing them in shuffled batches, which is what non-continual learning tools usually do. For the reader convenience, we rewrite the definition of real-time stream

that is used in this thesis. *y real-time stream* we intend a stream of data S where the interarrival times between a sample and the next one are always under a certain threshold Δ_S , or deadline as it is usually defined in real-time systems. Often, when considering online learning from a real-time stream, it is considered of most importance to keep the output stream O framerate at least equal to the input stream I framerate. This can be framed as $\text{fps}_O \geq \text{fps}_I$, where fps_O and fps_I are the output and the input stream framerates respectively. This is equivalent to $\frac{1}{\Delta_O} \geq \frac{1}{\Delta_I}$ which finally sets the threshold $\Delta_O \leq \Delta_I$. When considering sequential processing, Δ_O is almost always equal to the response time of a network. In the case of parallel processing, and in particular in the case of the paradigm proposed in this chapter, response time and Δ_O can actually be very different. Motivated by the intuitions behind existing libraries for batched data (Huang et al., 2018) and by approaches that rethink the neural network computational scheme making it local in time and along the network architecture (Marra et al., 2020; Tiezzi et al., 2022a; Betti and Gori, 2019; Betti et al., 2020a), we propose a different approach to *Pipeline Parallelism* specifically built for data sequentially streamed *over time*, where multiple devices work in parallel with the purpose of speeding-up computations. Considering D independent devices, such as D Graphics Processing Units (GPUs), the computational time of a feed-forward deep network empowered by the proposed approach theoretically reduces by a factor $1/D$. Noticeably, a D -layer network trained on a stream of data using our library has a theoretical framerate (or throughput) speedup of $D \times$ with respect to the case of a single device, thus computing D layers in the same time that is usually needed to compute only one. It is important to notice that Pipeline Parallelism does not decrease the response time of the network for a given input. Instead, it lowers the time between an output and the subsequent output, increasing the framerate of the output stream, outputting a stream that has the same framerate but delayed by the response time of the vanilla network. See section 6.2 for more details on this matter.

We experimentally show that the existing overheads due to data transfer among different devices are constant with respect to D in certain hardware configurations. On the reverse side, the higher throughput obtained by a pipeline parallelism are associated with a delay between the forward wave and the backward wave while they propagate through the network that is proportional to D , a feature that is not critical in applications in which data samples are non-i.i.d. and smoothly evolve over time. In applications that leverage the idea of parallelism, the user can select the optimal trade-off between speed and delays controlling D .

This chapter presents the following contributions to the use of Pipeline Parallelism to enable Lifelong Learning research. (1) We describe a software library named PARTIME (PARAllel processing over TIME)¹, written in Python and Py-

¹<https://github.com/sailab-code/partime>

Torch, specifically designed for efficient continual online learning in multi-GPU architectures using pipeline parallelism. Existing feed-forward multi-layer neural architecture can be easily embraced by our computational scheme, distributing inference and learning among multiple devices. (2) We leverage CUDA Streams² paired with CUDA Graphs³ to enable fast kernel scheduling, keeping a very low-level of abstraction of the parallel routines. We also provide automatic load balancing tools, specifically designed for the considered application context, and considering data transfer times in the temporal domain (differently by the offline case of (Huang et al., 2018)), as well as fast kernel scheduling by means of CUDA Graphs. (3) We experimentally evaluate the wall-clock running times with different numbers of devices, investigating the impact of the delays and of the data transfer overhead and comparing several architectures. In some configurations we get empirical speedups that are close to the theoretical ones. (4) We experimentally evaluate the quality of learning with different pipeline configurations in continual learning-inspired settings.

The rest of the chapter is organized as follows. In section 6.1 we describe the relationships of our work with existing literature and software. In section 6.2 we formally describe how computations are distributed over time, while section 6.3 is focused on their implementation and usage details. In section 6.4 we show experimental results and in section 6.5 we discuss limits and future extensions.

6.1 Related Work

The intuition of exploiting data- and/or model-level parallelism to speedup and scale training of deep networks (Yadan et al., 2013; Krizhevsky, 2014) has been largely followed in the last decade (Li et al., 2020; Rasley et al., 2020), also focusing on specific tasks or classes of neural nets (Shoeybi et al., 2019). The main motivation that inspired these approaches is the large memory requirements of recent neural models. Naively splitting the models into several components distributed across different devices hinders the overall resource utilization, with devices that are left idle waiting for other devices to complete their job. Pipeline parallelism deals with this under-utilization issue, not only splitting the network across multiple devices (e. g., the first layers in GPU 1, some of the following layers in GPU 2, etc.) but also splitting data into micro-batches and scheduling their forward propagation through the various *stages*, such that each device is left idle the least amount of time (Guan et al., 2019). Such computational paradigm yields an improvement on the number of inputs processed per seconds, based on the batch size, the number of stages and the communication overheads between devices. As a matter of fact, existing software libraries are designed for mini-batch mode processing, as

²<https://pytorch.org/docs/stable/generated/torch.cuda.Stream.html>

³<https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/>

they are designed to achieve parallelization over big dataset and stochastic gradient descent. Henceforth, we will assume that every implementation processes a mini-batch is then usually split into several micro-batches, that are identified by the order in which they enter the pipeline. In the following, we review the most popular pipeline parallelism-based libraries: GPipe (Huang et al., 2018), PipeDream and Pipedream-2BW (Narayanan et al., 2019, 2021).

GPipe (Huang et al., 2018) parallelizes the forward and backward phases across the micro-batches, with the latter being followed by a pipeline flush, where the accumulated gradients are used to update the weights, before moving to the next mini-batch. This induces a significant reduction of the maximum throughput, due to a *bubble* of idle time for most GPUs (see Figure 6.1a for a visual representation). On the other hand, it allows to faithfully replicate the gradients of a standard sequential processing, without the need to store a copy of the weights. GPipe needs to stash the activation values of each stage for all micro-batches to correctly compute gradients, incurring in a significant memory overhead. The usage of activation recomputation could alleviate this issue, but it further reduces the throughput by around 33%. See Figure 6.1a.

PipeDream (Narayanan et al., 2019) avoids the need of keeping the computations for the forward and backward stages completely separated. Forward and backward processing are interleaved continuously, without leaving any GPU idle. Weights are updated at each step so that, in order to compute the correct gradients for any batch, PipeDream adopts weights stashing in addition to activation stashing. The number of instances of the network weights that must be stored for a pipeline stage $s \in (1, \dots, D)$ is $D - s + 1$, being D the depth of the pipeline. Weights are updated at each step, but every input is guaranteed to be processed with the same set that already entered the pipeline will use the same set of stashed weights. PipeDream design allows for a significantly higher throughput with respect to GPipe, but incurs in a heavy memory overhead, which is not suitable for larger models. See Figure 6.1b.

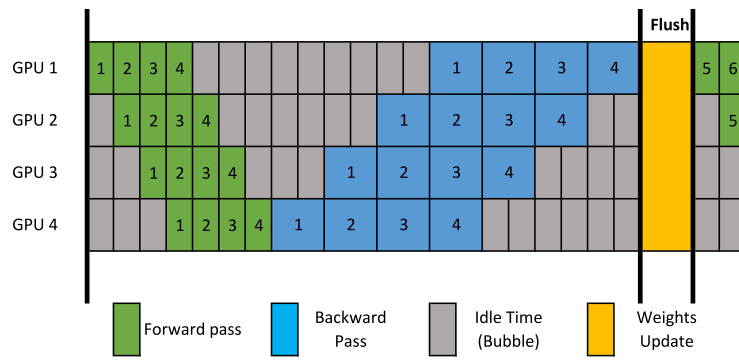
PipeDream-2BW (Narayanan et al., 2021) improves the PipeDream design by storing only two versions of the weights in each pipeline stage. A new weight version is generated, by applying the optimizer update rule, every $m \geq D$ micro-batches. However, the novel weight values are not used immediately to avoid inconsistencies in processing the data that already entered the pipeline. The new weights are therefore buffered and the oldest version is discarded. PipeDream-2BW introduces a delay between the weights that are used to compute the gradients and the weights that are actually updated. For a network f with weights $W^{(t)}$ for the t -th micro-batch, the gradient-based update rule can be expressed simply as $W^{(t+1)} = W^{(t)} - \nu \nabla f(W^{(t-1)})$ (where ν is the step-size). Thus, PipeDream-2BW tackles the memory overhead, significantly reducing it, and keeps a high speed-up of training times, at the cost of a small approximation of the gradients. See Fig-

ure 6.1c.

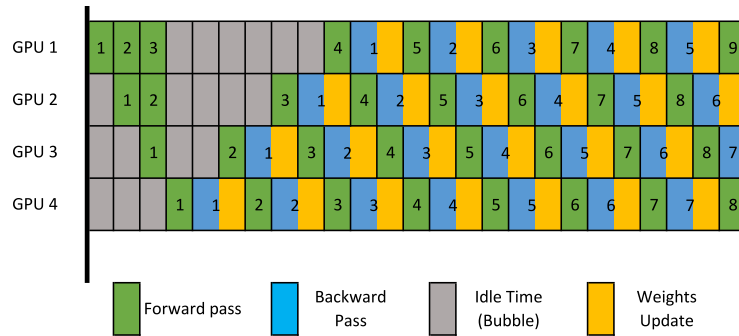
In the case of the proposed PARTIME, differently from all the existing approaches (to our best knowledge), we adapt the pipeline parallelism paradigm to the continual online learning scenario, where the input is provided in a sequential manner and there are no offline batches of data. PARTIME does not require mini-batches, and in real-world applications with real-time processing requirements, PARTIME can leverage multiple devices to achieve a processing frame rate closer or equal to the one of the input stream, while keeping the response time similar to that of the sequential network. The computation paradigm is related to that of PipeDream, relaxing the requirement of micro-batches that are now single examples from the input stream. Therefore, forward and backward phases are interleaved continuously without any pipeline flush, allowing a theoretical speed-up of the processing frame-rate of D . Differently from PipeDream(-2BW), PARTIME does not stash any activation or weights, further reducing the memory overhead, while keeping a significant speed-up. Differently from GPipe and both the PipeDream approaches, weights are updated at the end of the backward computation of each device, allowing continuous learning of the agent without any hiccup in the framerate, at the cost of an approximation of the gradients, that is not critical when the inputs slowly change over time. We depict in Figure 6.1d the PARTIME approach, where the numbers denote the input sample indices, while the sign - is used to indicate that the GPU is actually not idle (due to implementation constraints with CUDA Graphs, described in the rest of the paper), even if it using dummy data for the computations. See Figure 6.1d.

The theoretical groundings behind the approach implemented in PARTIME can be traced back to early studies on how to distribute the sequential computations of the network layers in a pipeline-like scheme (Petrowski et al., 1993). More recent activity (Betti and Gori, 2019; Marra et al., 2020) described how such a computational pipeline could be used to train networks where layers are independent modules interconnected by a special class of constraints, and that can learn in parallel. In (Betti and Gori, 2019) it is shown how different layers can propagate gradients when operating in a parallel manner, exploiting the temporal coherence of streamed data, and that is what we consider in this work. We remark that such a computational structure is also shared by the spatio-temporal local model of (Betti et al., 2020a), thus we note that PARTIME could naturally implement also alternatives to Backpropagation, even if this goes beyond the scope of this work.

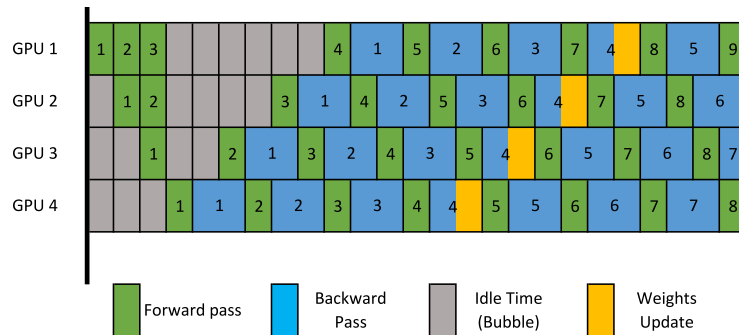
In the context of continual or Lifelong Learning, three main families of approaches have emerged in scientific literature: replay methods, regularization methods, parameter isolation methods—see (Delange et al., 2021) and references therein. In all these cases, a significant portion of the existing research focuses on a specific setting in the realm of computer vision, considering supervised problems (Rebuffi et al.,



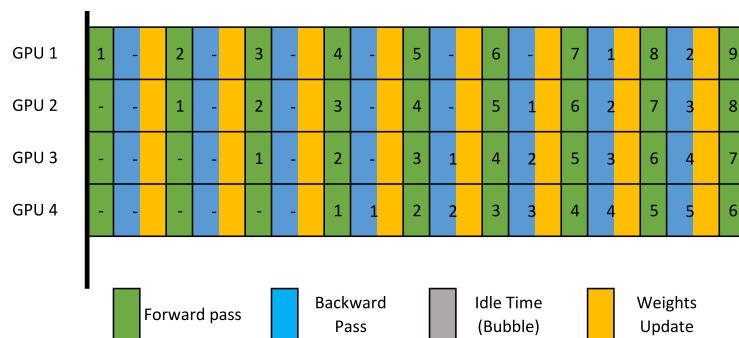
(a) GPipe (Huang et al., 2018)



(b) PipeDream (Narayanan et al., 2019)



(c) PipeDream-2BW (Narayanan et al., 2021)



(d) PARTIME (our contribution)

Figure 6.1: Processing timelines in different Pipeline implementations. Numbers indicate the ID of an input sample (or micro-batch).

2017; Gallardo et al., 2021) that sequentially introduce new tasks (e.g., new categories), with some recent attempts to operate in an unsupervised setting (Madaan et al., 2022). Despite the different features of all these approaches, they all intrinsically require to process a new input sample and compute the gradients of a loss function with respect to the network weights, that is what PARTIME implements.

6.2 Scalable and Parallel Local Computations Over Time

Let us consider a feed-forward neural network composed of L layers. We indicate with $\ell_i(z, w_i)$ the function computed by the i -th layer when the input z is provided, while w_i is the set of the learnable parameters involved in the layer-internal operations. If $f(x, \omega)$ is the function computed by the network when the input x is provided, being $\omega = \cup_{i=1}^L w_i$, we can formally describe f as

$$f(x, \omega) = (\ell_L(\cdot, w_L) \circ \ell_{L-1}(\cdot, w_{L-1}) \circ \dots \circ \ell_1(\cdot, w_1))(x). \quad (6.1)$$

Given a scalar loss function \mathcal{L} that depends on the net outputs and other eventually available information γ (e.g., supervisions), the Backpropagation algorithm provides a clever way of computing $\nabla_{w_i} \mathcal{L}$, the gradient of \mathcal{L} with respect to some w_i , exploiting the composite structure of f . Formally, if we indicate with o^j the output of the j -th layer,

$$\begin{aligned} \nabla_{w_i} \mathcal{L}(f(x, \omega), \gamma) &= \frac{\partial \mathcal{L}(o^L, \gamma)}{\partial o^L} \cdot \frac{\ell_L(o^{L-1}, w_L)}{\partial w_i} \\ &= \frac{\partial \mathcal{L}(o^L, \gamma)}{\partial o^L} \cdot \frac{\partial \ell_L(o^{L-1}, w_L)}{\partial o^{L-1}} \cdot \frac{\partial \ell_{L-1}(o^{L-2}, w_{L-1})}{\partial o^{L-2}} \\ &\quad \dots \cdot \frac{\partial \ell_i(o^{i-1}, w_i)}{\partial w_i}, \end{aligned} \quad (6.2)$$

where \cdot is the operator involved in the classic chain-rule.⁴ Let us assume that layers are divided into $D \leq L$ non-overlapping sets also referred to as *stages*, where the h -th stage collects consecutive layers with indices in $[a_h, b_h]$, $1 \leq a_h \leq b_h \leq L$, and $a_{h+1} = b_h$. We indicate with p_h the h -th stage, involving layer functions $\ell_{a_h}, \ell_{a_h+1}, \dots, \ell_{b_h}$. We also assume that $p_{h+1}^{a_{h+1}:b_{h+1}}$ is such that $a_{h+1} = b_h + 1$, thus, in what follows we will drop the superscripts unless explicitly needed. We indicate with ω_h the set that collects all the weights belonging to the layers in the h -th stage, $\omega_h = \cup_{i=a_h}^{b_h} w_i$. The input/output of p_h are o_{a_h} and o_{b_h} , respectively. Overloading the notation p_h to

⁴It is a product when dealing with scalar quantities, otherwise it is the operator that appropriately combines Jacobian matrices or tensors.

also refer to the function computed by the h -th stage, for simplicity, we can rewrite Equation 6.1 as

$$f(x, \omega) = (p_D(\cdot, \omega_D) \circ p_{D-1}(\cdot, \omega_{D-1}) \circ \dots \circ p_1(\cdot, \omega_1))(x). \quad (6.3)$$

Let us assume w_i to belong to the weights ω_h of stage p_h , i.e., $b_h \leq i \leq a_h$. Recalling that $o_{a_h} = o_{b_{h-1}}$, for all valid h , the gradient $\nabla_{w_i} \mathcal{L}$ is then

$$\begin{aligned} \nabla_{w_i} \mathcal{L}(f(x, \omega), \gamma) &= \frac{\partial \mathcal{L}(o_{b_D}, \gamma)}{\partial o_{b_D}} \cdot \frac{p_D(o_{a_D}, \omega_D)}{\partial o_{b_h}} \cdot \frac{p_h(o_{a_h}, \omega_h)}{\partial w_i} \\ &= \frac{\partial \mathcal{L}(o_{b_D}, \gamma)}{\partial o_{b_D}} \cdot \frac{\partial p_D(o_{a_D}, \omega_D)}{\partial o_{a_D}} \\ &\quad \cdot \frac{\partial p_{D-1}(o_{a_{D-1}}, \omega_{D-1})}{\partial o_{a_{D-1}}} \\ &\quad \dots \cdot \frac{\partial p_h(o_{a_h}, \omega_h)}{\partial w_i}, \end{aligned} \quad (6.4)$$

where the last term, $\partial p_h(o_{a_h}, \omega_h) / \partial w_i$, can be computed by backpropagating the signal inside the h -th stage,

$$\begin{aligned} \frac{\partial p_h(o_{a_h}, \omega_h)}{\partial w_i} &= \frac{\partial \ell_{b_h}(o^{b_h-1}, w_{b_h})}{\partial o^{b_h-1}} \cdot \frac{\partial \ell_{b_h-1}(o^{b_h-2}, w_{b_h-1})}{\partial o^{b_h-2}} \\ &\quad \dots \cdot \frac{\partial \ell_i(o^{i-1}, w_i)}{\partial w_i}. \end{aligned} \quad (6.5)$$

Whenever data is provided by a generic source stream \mathcal{S} , at each time instant t a new sample $x^{(t)}$ is made available, and the following sample is provided after $\Delta_{\mathcal{S}}^{(t)}$ seconds. Without any loss of generality, we will consider the case in which $\Delta_{\mathcal{S}}^{(t)}$ is constant $\forall t$, thus we will just use the notation $\Delta_{\mathcal{S}}$. For example, in the case of a video stream, $\Delta_{\mathcal{S}} = 1/\text{FPS}$, being FPS the frame rate of the video (constant rate). Processing a data sample $x^{(t)}$ requires computing $f(x^{(t)}, \omega)$ and, in a Lifelong Learning horizon, also $\nabla_{w_i} \mathcal{L}(f(x^{(t)}, \omega), \gamma)$, for all the valid i 's (plus the weight update operations), that is not an instantaneous processing, especially in very deep networks or when the input size is large. We indicate with $\Delta_{\mathcal{A}}$ such amount of processing time. Whenever $\Delta_{\mathcal{A}} > \Delta_{\mathcal{S}}$, real-time learning is not possible anymore, requiring to skip some frames, since buffering data would just create a constantly increasing queue whose processing is unfeasible. Skipping frames would result in an inherent loss of information that could limit the learning capabilities of the network. As a simple example, consider a network that learns by enforcing motion coherence over consecutive frames (Betti et al., 2020c). If there is too much distance (in time) between the frames of the pair, there might be poor correlation between them, thus making learning not feasible. Of course, distributing the computations over multiple devices is beneficial for networks that do not fit the memory of a single one, but

processing time is still limited by the intrinsic sequential nature of the layer-wise or stage-wise computations, so that a device must wait the previous one to finish its job before starting its own activity.

In this paper we focus on an alternative way of organizing computations over time that is meant for hardware solutions equipped with D computational devices, in particular GPUs, each of them dedicated to the computations of a single stage p_h . Before going into further details, we remark that what we propose is generic and essentially holds also for other types of devices that can work in parallel or in case of devices that include multiple parallel computational units. While a single GPU actually belongs to the latter category, its parallel computation capabilities are aimed at speeding up lower-level operations (matrix multiplications, convolutions, ...) that usually exploit most of the GPU resources (e.g., CUDA blocks in NVIDIA cards (Cheng et al., 2014)). As a result, a single GPU is not well-suited for speeding up, for example, multiple stages, even if scheduled for parallel execution.⁵ Let us assume that layer stages are created in a way to have very similar computational times for each stage in the target hardware. Hence, a device that process a single stage in $\Delta_{\mathcal{P}}$ seconds will require $\Delta_{\mathcal{A}} = D\Delta_{\mathcal{P}}$ seconds to compute $f(x^{(t)}, \omega)$, both in the case in which all the stages are sequentially executed on a single device, or if each stage is processed on an independent device, ignoring data transfer overheads. Conversely, a model equipped with Pipeline Parallelism is ready to process another sample whenever the first GPU has done processing the first stage, that only takes $\Delta_{\mathcal{P}}$, while the vanilla model takes $D\Delta_{\mathcal{P}}$ before being ready to process another sample from the stream. This implies that a pipeline scheme can real-time process the streamed data if $\Delta_{\mathcal{P}} \leq \Delta_{\mathcal{S}}$, instead of $\Delta_{\mathcal{A}} = D\Delta_{\mathcal{P}} \leq \Delta_{\mathcal{S}}$, increasing the throughput by a factor D . In other terms, the highest framerate a vanilla network can reach is $\frac{1}{\Delta_{\mathcal{A}}} = \frac{1}{D\Delta_{\mathcal{P}}}$, instead a pipelined network can reach $\frac{1}{\Delta_{\mathcal{P}}}$, which is D times the framerate. This allows the pipelined network to process the online stream at real-time framerates even if the vanilla network has a processing framerate under D times the time between subsequent frames in the input stream. The delay introduced to the output stream with respect to the input stream is fixed and dependent on the response time of the vanilla network. To reduce the response time, other parallelism techniques can be introduced in conjunction with Pipeline Parallelism, which we will show is a general purpose paradigm that can be easily applied to sequential models to increase the framerate in settings where the response time is not important, but tracking the framerate is, as is the case for processing an online video stream without strict response time requirements.

⁵In practice, scheduling multiple stages in the same GPU for a potentially parallel execution boils down to the almost serial execution of them, since the lower-level operations within each stage exploit most of the GPU resources. In our experience, there is just a narrow set of cases in which this can lead to non-negligible speed-ups.

As in classic pipeline parallelism, the h -th stage of the pipeline performs the forward and backward phases of the layers of p_h , using the output that was provided by stage $h - 1$ at the previous time step, and the gradients provided by stage $h + 1$. In PARTIME, each stage also updates the weight values right after the backward computation has ended (Figure 6.1d). Thus, it is necessary to include an explicit time dependence in the set of weights characterizing each stage $\omega_i^{(t)}$. Then, the output of stage i at time t will be described by the temporal index on the values of the weights ($\omega_i^{(t)}$), the output of stage h at time t is described as,

$$\begin{aligned} o_{b_h}^{(t)} &= p_h(o_{b_{h-1}}^{(t-\Delta p)}, \omega_h^{(t)}) \\ &= \left(p_h(\cdot, \omega_h^{(t)}) \circ \dots \circ p_1(\cdot, \omega_1^{(t-(h-1)\Delta p}) \right) (x^{(t-(h-1)\Delta p)}), \end{aligned} \quad (6.6)$$

According to the above equation, the network output at time t is given by $o_{b_D}^{(t)} = p_D(o_{b_{D-1}}^{(t-1)}, \omega_D^{(t)})$, and it is equal to

$$\begin{aligned} &f(x^{(t-(D-1)\Delta p)}, \{\omega_D^{(t)}, \omega_{D-1}^{(t-\Delta p)}, \dots, \omega_1^{(t-(D-1)\Delta p)}\}) \\ &= \left(p_D(\cdot, \omega_D^{(t)}) \circ \dots \circ p_1(\cdot, \omega_1^{(t-(D-1)\Delta p}) \right) (x^{(t-(D-1)\Delta p)}), \end{aligned} \quad (6.7)$$

that is essentially the analogous of (6.3) when making explicit the different time indices attached to the weights involved in processing a sample from the input stream. Of course, as in every pipeline-based model, the output associated to each sample becomes available with a delay that is $D\Delta p$, not a crucial issue in case of smoothly evolving streams with relatively large frame rates, as the ones we consider in this paper.

The backward propagation in PARTIME follows a similar approach to the forward case, with gradients propagating through the pipeline from the loss function down to the first stage. Assuming w_i to belong to the weights ω_h of stage p_h we generalize (6.4) to

$$\begin{aligned} (\nabla_{w_i} \mathcal{L})^{(t)} &= \frac{\partial \mathcal{L}(o_{b_D}, \gamma^{(t-(D-h)\Delta p})}{\partial o_{b_D}} \Bigg|_{o_{b_D}^{(t-(D-h)\Delta p)}} \\ &\quad \cdot \frac{\partial p_D(o_{b_{D-1}}, \omega_D^{(t-(D-h)\Delta p})}{\partial o_{b_{D-1}}} \Bigg|_{o_{b_{D-1}}^{(t-(D-h+1)\Delta p)}} \\ &\quad \dots \frac{\partial p_{h+1}(o_{b_h}, \omega_{h+1}^{(t-\Delta p)})}{\partial o_{b_h}} \Bigg|_{o_{b_h}^{(t)}} \cdot \frac{\partial p_h(o_{b_{h-1}}, \omega_h)}{\partial w_i} \Bigg|_{w_i^{(t)}}. \end{aligned} \quad (6.8)$$

The last term $\partial p_h(o_{b_{h-1}}, \omega_h) / \partial w_i|_{w_i^{(t)}}$ is evaluated by means of a backpropagation of the signal within the given stage h according to (6.5), while the product of

the previous terms is propagated through the stages. Notice that, focusing on a specific stage h , the PARTIME computational structure is characterized by a delay consisting in $2(D - h)$ steps in-between the corresponding forward and backward “waves”, as can be seen in Figure 6.1d, checking each stage/GPU-line and counting the steps between data with the same index—recall that each step consists of forward, backward, update. This leads to an approximation in the evaluation of the gradients since, during such a time interval, the input of the stages changes. Of course, the slower the input stream is varying, the less impacting is the approximation. The delay is zero for $h = D$ since, in this case, the last stage computes the forward, backward (and update) operations related to a given input at the same time. This also means that, for the last stage, the computation of the gradients are not influenced by this delay. However, it is not only a matter of changing the stage input, since a similar consideration holds for the values of weights, that get updated (thus they change) at each computational step, as we already anticipated in Equation 6.7. Weights get updated at each computational step and this will play a role in the evaluation of the gradients.

Even if this will play a role in the evaluation of the gradients, a small learning rate can mitigate abrupt changes in the values of the weights, making the resulting approximation more appropriate. Notice that this second source of approximation concerns also the last layer: even if the forward and backward computation are referring to the same input of the stage, the forward propagation on the stages below involved weights at different time instants (as was shown in Equation 6.7). In the next section we will describe the implementation of the ideas here presented. To this purpose, let us note that we can also write (6.8) as

$$(\nabla_{w_i} \mathcal{L})^{(t)} = g_h^{(t)} \cdot \left. \frac{\partial p_h(o_{b_{h-1}}^{(t-\Delta p)}, \omega_h)}{\partial w_i} \right|_{w_i^{(t)}}, \quad (6.9)$$

where we have contextually defined

$$\begin{aligned} g_h^{(t)} = & \left. \frac{\partial \mathcal{L}(o_{b_D}, \gamma^{(t-(D-h)\Delta p})}{\partial o_{b_D}} \right|_{o_{b_D}^{(t-(D-h)\Delta p)}} \\ & \cdot \left. \frac{\partial p_D(o_{b_{D-1}}, \omega_D^{(t-(D-h)\Delta p})}{\partial o_{b_{D-1}}} \right|_{o_{b_{D-1}}^{(t-(D-h+1)\Delta p)}} \\ & \dots \left. \frac{\partial p_{h+1}(o_{b_h}, \omega_{h+1}^{(t-\Delta p)})}{\partial o_{b_h}} \right|_{o_{b_h}^{(t)}} \end{aligned} \quad (6.10)$$

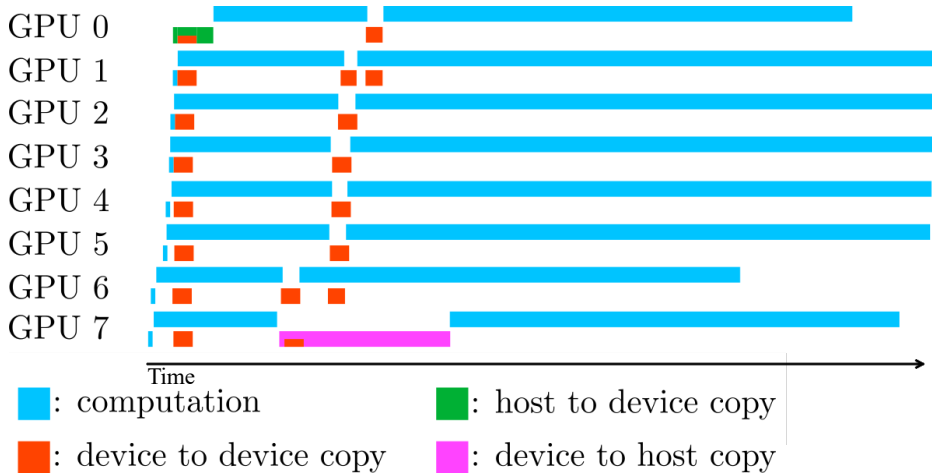


Figure 6.2: Capture of a single forward/backward step of PARTIME. With the use of CUDA Graphs, every GPU starts processing almost at the same time, reducing communications overhead at its minimum. The result is an almost 8x speed-up.

that, starting from $g_D^{(t)} = \partial \mathcal{L}(o_{b_D}, \gamma^{(t)}) / \partial o_{b_D} |_{o_{b_D}^{(t)}}$, can be also expressed through to the following recursive relation

$$g_h^{(t)} = g_{h+1}^{(t-\Delta p)} \cdot \frac{\partial p_{h+1}(o_{b_h}, \omega_{h+1}^{(t-\Delta p)})}{\partial o_{b_h}} \Big|_{o_{b_h}^{(t)}}. \quad (6.11)$$

6.3 PARTIME

The PARTIME software library is written in PyTorch⁶ and it expects the user to provide a classic network of type `torch.nn.Sequential`, that is automatically converted into a format that will enable pipelined computations, both in the forward and backward phases. Despite the general formulation of the ideas behind PARTIME, the current version of the library leverages CUDA-based facilities, in particular CUDA Streams and CUDA Graphs, to setup the parallel execution scheme, thus it is designed for NVIDIA GPUs. In algorithm 1 we provide a high-level description of the operations performed by each stage/device.

In order to activate the concurrent execution of independent queues of GPU tasks, PARTIME creates multiple CUDA Streams on each device. Each stream holds a queue of sequential tasks which are executed in-order, while the different streams interleave their tasks in no specified order. Since CUDA Streams are handled in an asynchronous manner (i.e., they do not wait the results of each enqueued task, even if such results are used in the rest of the code), PARTIME relies on specifically

⁶We exploited PyTorch 1.11 for our implementation.

placed “events” (provided by the CUDA APIs) to ensure proper synchronization of the different streams. An *Event* can be placed on the Stream queue to synchronize Streams on it, which will stop their processing until that Event has popped out of the queue. A major drawback of a vanilla CUDA Stream-based implementation is the

- 1: **if** 1st stage **then**
- 2: Copy $x^{(t)}$ from host memory to the *stable* input memory area.
- 3: **else**
- 4: Copy *temporary* memory area to *stable* input memory area.
- 5: **end if**
- 6: Start forward step, using the *stable* input memory {Eq. (6.6)}.
- 7: **if** last stage **then**
- 8: When 6 completes, copy last stage output to host memory.
- 9: **else**
- 10: Copy gradients from the next stage $\{g_{h+1}^{(t-\Delta p)}\}$ in Eq. (6.11)
- 11: When 6 completes, copy output to the next-stage *temporary* input memory area (to avoid overwriting *stable* data—currently used by the next-stage).
- 12: **end if**
- 13: Compute stage-related gradients {Eq. (6.9)}
- 14: Update weights.

Algorithm 1: Operations performed by each device/stage. Each stage expects input data to be stored in the so-called *stable* memory area, and it also has the use of a *temporary* one.

communication-overhead introduced when the CPU enqueues a task to the GPU streams. Such overhead quickly becomes negligible when dealing with tasks that run for a long time, but not when processing streams of data with the purpose of splitting computations into fast-processing parallel stages, where (in some configurations) enqueueing could take more time than the actual execution of each stage. The overhead is crucial to achieve maximum parallelization speed-up. PARTIME bypasses this issue exploiting the recently introduced NVIDIA CUDA Graphs, that are able to handle the dependencies among the various GPU tasks in a warmup stage, generating a compact Directed Acyclic Graph (DAG) that summarizes all the operations. This makes all the GPU-task virtually collapse into a single one, with a scheduling overhead that is paid only once for all the wrapped tasks. Since CUDA Graphs might not be supported by older GPUs, PARTIME can easily switch-back to the case in which they are not used.

In Figure 6.2 we report the outcome of an operation-capture performed with the NVIDIA NSIGHT System, of a single pipeline step, confirming that all the GPUs are able to work in parallel, distributing the computational burden in an effective manner. On the flip side, CUDA Graphs only deal with statically allocated data and do not support some PyTorch operations. A Graph operates on the same memory areas each time it is replayed, meaning that buffers for input/output must be statically

created at the creation of the pipeline, and the Graph will change those buffers in an inplace fashion. Of course, in order to get the most out of the pipeline scheme, the computational time of the different stages should be comparable, to avoid the slower stage to reduce the throughput. PARTIME provides a stage balancing procedure, in which the network layers are automatically partitioned, that is what we used both in Figure 6.2 and in the following experiments of this paper. First of all, the copy-times from/to host memory to GPUs are measured, assigning the first/last pipeline stages to the fastest GPUs (the speed depends on the hardware configuration of the GPUs in the host machine). Then, PARTIME evaluates the processing times of all the layers in each GPU, and also the GPU-to-GPU layer-output transfer times and then it splits layers into stages trying to make their overall computational times similar, under some constraints, with the same algorithm exploited in related libraries (Kim et al., 2020) that, differently from PARTIME, do not consider data-transfer times.

In the PARTIME library, a `Pipeline` object holds an ordered list of `Stages`. Each of the `Stages` contains a non-overlapping portion of the original network and it also manages the input and output buffers for that part of the network. This means that processing an input through the list of `Stages` in order will produce the same output as the original network with around the same processing time. The `Pipeline` is initialized using (i) the neural network model to be handled, (ii) the number of stages (eventually, how layers should be distributed), (iii) the list of available GPU devices (for maximum performance, it should be the same as the number of stages, but it can be smaller, making multiple stages execute on the same device), (iv) the optimizer settings for weight updates (eventually, a user defined loss functions), and (v) a sample tensor whose shape is leveraged to infer the shapes of static memory allocations, required by CUDA Graphs. The PARTIME routines splits the original network into the required stages and transfer them to their respective GPU, eventually generating a CUDA Graph if requested by the user.

`Pipeline` objects provide a `forward` method that processes inputs with the same shape of the previously described sample tensor, triggering the computations described in algorithm 1 over all the `Stages` (forward, backward, update). A `forward` call computes a single step of the pipeline: each stage processes its input and copies its output to the following stage' input. Eventually, the user could decide to advance the pipeline by one step without providing new inputs. In that case, the first stage will process the same input as in the last step. Every other stage is executed even if not holding data provided by the user. At the end of the step, the `forward` method returns the content of the last stage' output buffer.

In the following we provide a simple code snippet with an example of usage of PARTIME, showing that a few lines of code are needed to setup a pipeline-based execution.

```

from partime.pipeline import Pipeline
from partime.balancing import balance_pipeline_partitions

net = YourSequentialModel()
balance = balance_pipeline_partitions(net,
    devices, len(devices))

pipeline = Pipeline(
    net,          # Network to be wrapped
    sample_input, # A tensor with the
                 # same shape as the input stream
    balance,      # A list that determines how
                 # layers are split, e.g. [8, 10, 12, 11]
    devices,      # List of devices to use
    cuda_graph,   # Flag to enable CUDA Graph
    loss_fn,      # A callable that returns the loss
    sample_target, # A tensor with the
                 # same shape as the targets
    optim_settings # Tuple with optimizer class
                 # and dict of hyper-parameters
)

for idx, (inp, target) in enumerate(stream):
    pipeline.forward(inp, target)
    if idx < len(pipeline.stages) - 1:
        continue # The first input still has not reached
                 # the end of the pipeline
    else:
        outputs = pipeline.outputs_buffer
        loss = pipeline.loss_buffer
        # Print/display/log loss and output

```

The method `pipeline.forward(inp, target)` enqueues the input into the pipeline and execute a forward/backward step. The output and loss buffers are filled with the output of the last stage and the output of the provided loss function, respectively. Notice that the first `len(pipeline.stages)-1` steps are needed to make the first input propagation reach the last stage (see code above), and the respective outputs/loss values must be neglected.

6.4 Experiments

We performed several experiments to showcase the processing speedup obtained via the PARTIME library, starting with a stream of visual data, composed of 10000 frames. We evaluated the speed at which the network performs *inference* (forward only), comparing it to the time needed for the same operations with vanilla sequential (i.e., non-pipeline-based) models. Moreover, we also considered the case in which the PARTIME pipeline is used to wrap a neural model that performs continual online *learning*, thus also including the backward and update steps. Then, we evaluated the quality of the trained model in a continual online motion estimation

experiment and in a classification problem based on a stream of images from the CIFAR-10 dataset, providing the network with data taken from a sliding window of samples, to avoid abrupt changes in input at consecutive time steps. We note that PARTIME increases the maximum framerate of the network, but the response time (i.e. the time between an input and its respective output becoming available) is not reduced - as typical of any pipelining scheme. A shorter response time could in principle be achieved by integrating different parallel techniques with the pipeline parallelism, but this is out of the scope of this work.

A. Pixel-wise Predictions. Our first experimental activity is about a neural architecture composed by 150 convolutional layers having fixed input/output resolution (i.e. no pooling or stride > 1), thus simulating the prediction of pixel-wise features. We assume to deal with input tensors having a squared spatial resolution of $R \times R$ pixels, with $R \in \{256, 1024\}$ and 3 channels. In bold measurements greater than 50% than the theoretical speed-up. Table 6.1 (top) shows the speedup in *inference*, averaged over the considered forward steps, reporting in bold those speedups that are greater than 80% of the theoretical speedup. We considered different settings, consisting in various combinations of input resolutions ($R \times R$) and number of output features ($F \in \{1, 10, 100\}$) to better evaluate the impact of the communication overheads with different data sizes. We denoted each setting with the compact notation R/F . We also considered varying numbers of pipeline stages ($\#STAGES \in \{2, 4, 8\}$). The obtained results confirm the huge contribution of CUDA Graph in the low-resolution settings, where communications overheads are more impactful than computational times, while advantages of CUDA Graph are less evident with bigger input resolutions. In all the cases PARTIME provides significant improvements over the vanilla sequential network, reaching results closer to the theoretical case. Noticeably, with $R = 1024$ the exploitation of PARTIME yields a speedup $\approx \times 7$, close to the maximum theoretical speed-up of 8. In Table 6.1 (bottom) we report the *learning* case, in which also the backward phase and optimization step are included. Speedups are even greater than the inference case, as backward computations increase the computational cost of each single pipeline stage, reducing the relative impact of data-transfer overhead (see Figure 6.2).

B. Image Classification. The second experimental activity is about neural architectures composed of 150 convolutional layers interleaved with pooling layers, with the final output pooled to a vector with F elements, the number of output classes (1 vector per image). The main goal of this activity is to evaluate potential pipeline balancing issues due to the different spatial resolutions of the layers (i.e. layers processing inputs with spatial resolution smaller than R have less impact in the overall computation). Table 6.2 (top, same structure of Table 6.1, reporting in bold the cases in which the speedups are greater than half of the theoretical one) shows that CUDA Graph yields the best results even in this experience. The aforementioned

Table 6.1: Pixel-wise predictions. Speed-up achieved by PARTIME considering only inference (top) and both forward and backward learning phases (bottom), varying the input tensor resolution R and the number of output channels F (rows), as well as the number of pipeline stages (columns), and investigating the advantages of CUDA Graph. In bold measurements greater than 80% than the theoretical speed-up.

		Without CUDA Graph			With CUDA Graph		
		2	4	8	2	4	8
#STAGES R / F							
Inference Only	256 / 1	1.12	1.17	1.15	1.89	3.74	6.41
	256 / 10	1.09	1.12	1.10	1.68	3.65	6.46
	256 / 100	1.08	1.11	1.08	1.72	3.13	5.05
	1024 / 1	1.82	3.28	5.27	1.86	3.87	7.14
	1024 / 10	1.82	3.21	5.53	1.88	3.57	6.65
	1024 / 100	1.72	2.79	4.44	1.69	2.99	4.81
Learning	256 / 1	1.36	1.45	1.46	1.96	3.80	7.16
	256 / 10	1.38	1.42	1.48	1.99	4.11	7.45
	256 / 100	1.28	1.46	1.48	1.99	3.57	6.70
	1024 / 1	1.81	3.43	5.63	1.95	3.66	6.61
	1024 / 10	1.85	3.63	6.26	1.95	3.81	7.37
	1024 / 100	1.88	3.39	5.57	1.89	3.57	6.35

layer/stage balancing issue causes an under-utilization of some of the GPUs, that is more evident in the case of $R = 1024$, where the maximum speed-up is almost $\times 3$ even with 8 GPUs.

C. Image Classification with Residual Connections. We performed another image classification experience using a ResNet-152 architecture, customizing the final classification head to yield F output classes/features. Skip connections from layer i to layer $j > i$ are propagated through all the stages in-between i and j , by means of identity mappings. Therefore, we remark that skip connections introduce further copy operations between GPU devices. Nonetheless, the speed-up achieved by PARTIME are similar to the previous experience, suggesting that CUDA Graph is still very helpful in mitigating performance losses due to communications overheads. Settings with $\#STAGES \in \{2, 4\}$ get closer to the theoretical speed-up, since the data that is about the skip connections need to be propagated through less stages.

D. Continual Online Image Classification. In our next experience we simulated a continual online learning process where a neural model is trained on the CIFAR-10 dataset, with examples provided in a sequential manner. The training procedure is inspired to Replay Methods (see section 2.2) and it collects inputs using a sliding window to build up a “replay batch”, which is then fed to the pipeline. Each time

Table 6.2: Image Classification without and with residual connections (top and bottom part of the table, respectively), varying the input tensor resolution R and the number of output channels F (rows), as well as the number of pipeline stages (columns), and investigating the advantages of CUDA Graph.

		Without CUDA Graph			With CUDA Graph		
		2	4	8	2	4	8
#STAGES R / F							
Classifier	256 / 1	1.16	1.13	1.06	0.78	2.59	4.31
	256 / 10	1.11	1.04	1.00	0.99	2.55	3.48
	256 / 100	1.10	1.06	1.00	1.69	2.67	4.39
	1024 / 1	1.63	1.67	1.18	1.43	2.61	2.89
	1024 / 10	1.66	1.64	1.38	0.80	1.64	2.52
	1024 / 100	1.43	1.63	1.49	1.20	1.24	2.79
ResNet	256 / 1	1.10	1.07	1.04	1.40	2.94	2.11
	256 / 10	1.11	1.04	1.02	1.77	2.99	4.50
	256 / 100	1.11	1.06	0.98	1.66	2.93	4.31
	1024 / 1	1.24	1.23	0.99	1.42	2.34	1.50
	1024 / 10	1.23	1.20	1.15	1.07	2.72	3.88
	1024 / 100	1.27	1.22	1.14	1.69	3.28	2.94

a new sample is provided, the oldest input in the replay batch is replaced by the new one, while the rest is kept as is. This ensures that the PARTIME assumption of processing slowly-varying inputs is approximately met. The learning rate is also chosen accordingly to avoid changing the weights too much between each step. We considered a ResNet-50 with 10-classes classification head, trained using the Adam optimizer with learning rate $\mu = 0.001$, streaming the whole dataset 5 times, testing different sliding-window/batch sizes in $\{64, 256, 1024\}$. Figure 6.3 compares the accuracy on the test data for the different batch sizes. As expected from the theoretical analysis, there is a trade-off between performances on the task and increased training speed. With respect to training the sequential network, training a 2-stage pipelined network takes around half of the time of the sequential model. The results show that when increasing the batch size, the pipeline is able to learn better predictors, as we better implement the slowly-changing input condition.

We can look more closely at the behavior of the learning trajectory when using pipeline parallelism by observing Figure 6.4 and Figure 6.5, where it is respectively plotted the evolution in time of accuracy and loss, respectively. In Figure 6.4a and Figure 6.4b we see, the evolution of accuracy for batch size set to, respectively, 64 and 1024. The evolution of loss for batch sizes of 64 and 1024 is similarly reported respectively in Figure 6.5a and Figure 6.5b Interestingly, for batch size equal to 64, the accuracy for the pipelined network evolves similarly to the case without any

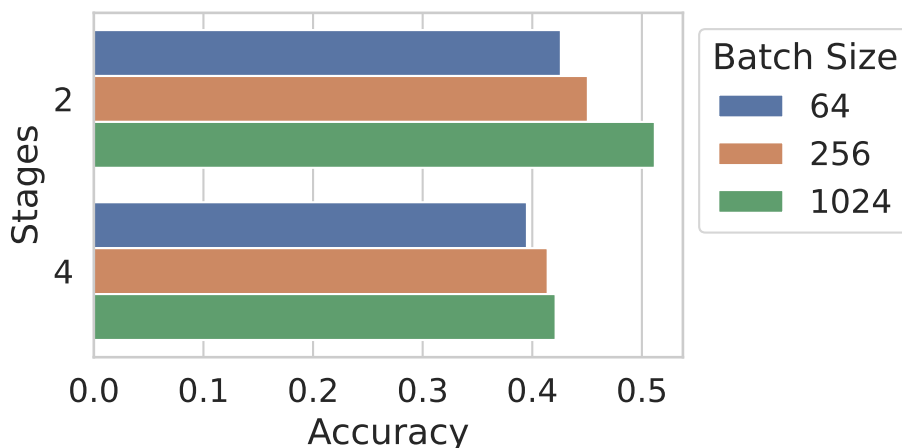
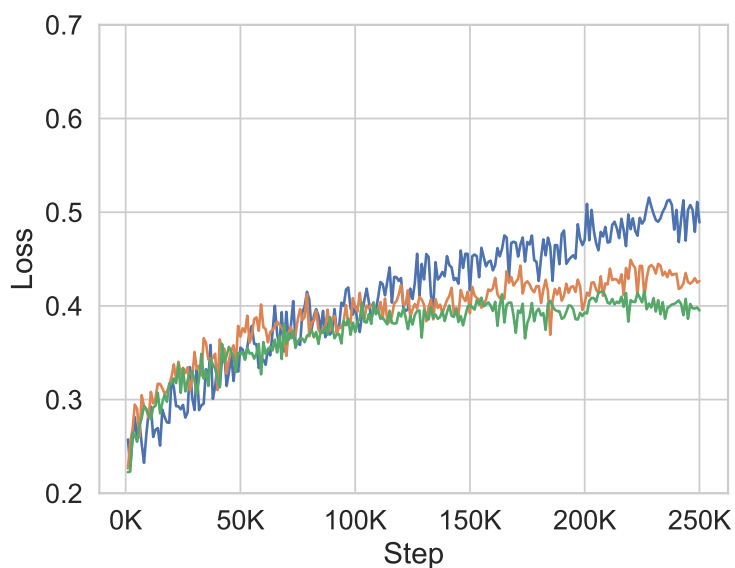


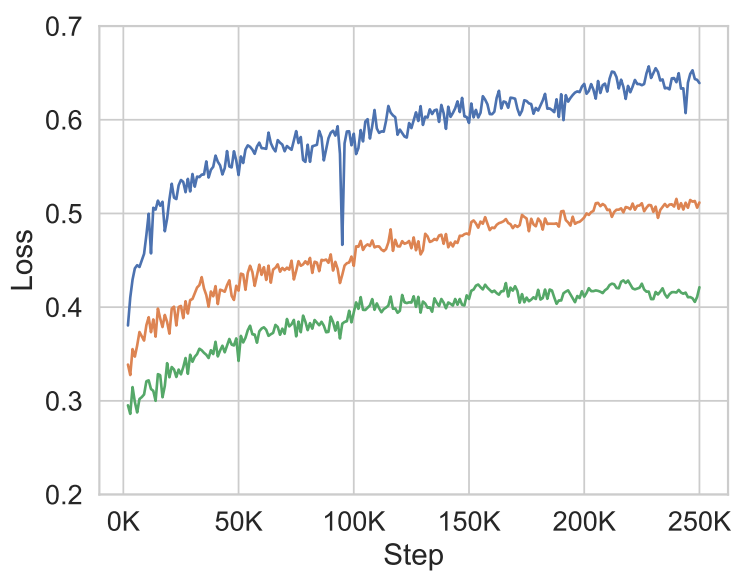
Figure 6.3: Test accuracy in the continual online image classification described in the main text. For reference, the baseline accuracies for a sequential network trained without pipelining are 0.489, 0.553, and 0.639 for batch sizes of 64, 256 and 1024, respectively.

pipeline. Conversely, in the case of 1024, the trajectories are cleanly separated. In the case of the loss we see a similar trend for both batch sizes, but it is interesting to note that while having a higher final loss, the trajectory of the metric over time is far more stable and with less high fluctuations, when compared to the trajectory of the non-pipelined network, even without any big fluctuations in the case of batch size equal to 1024. The rise of the loss metric associated with a plateau in the accuracy could indicate a case of overfitting on the data, where the delay in the gradients impede the network from extracting anymore useful data. While it is clear in Figure 6.3 that increasing the batch size better approximates the assumption of slowly changing inputs, it also advantages the non-pipelined network that was designed to better work with great batches of data. This is also evident seeing that batch sizes of 64 are less of an advantage for common batch-training networks in terms of accuracy and loss, while still keeping a significant disadvantage in terms of training time. Nonetheless, it is important to keep in mind that the training of the pipelined networks are, respectively, almost a half and a fourth, bringing forth a great advantage in terms of time that allows the researcher to tweak the model and find more suitable architectures (see Betti et al. (2020a)).

E. Continual Online Optical Flow Estimation. Our last experimental activity fits more closely into the idea of learning agents that perceive a visual stream and continuously learn sample by sample, in this case considering the optical flow learning problem (Brox et al., 2004), which consists of estimating the displacement field for all the pixels in a given frame pair. Such problem has been extensively investigated with the aid of deep neural networks, and we replicated the experience of (Marullo

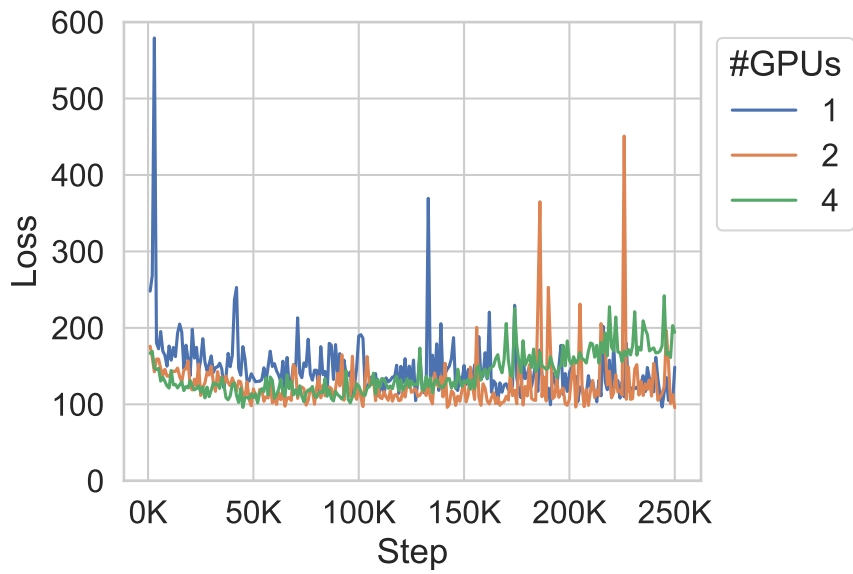


(a) Batch Size = 64

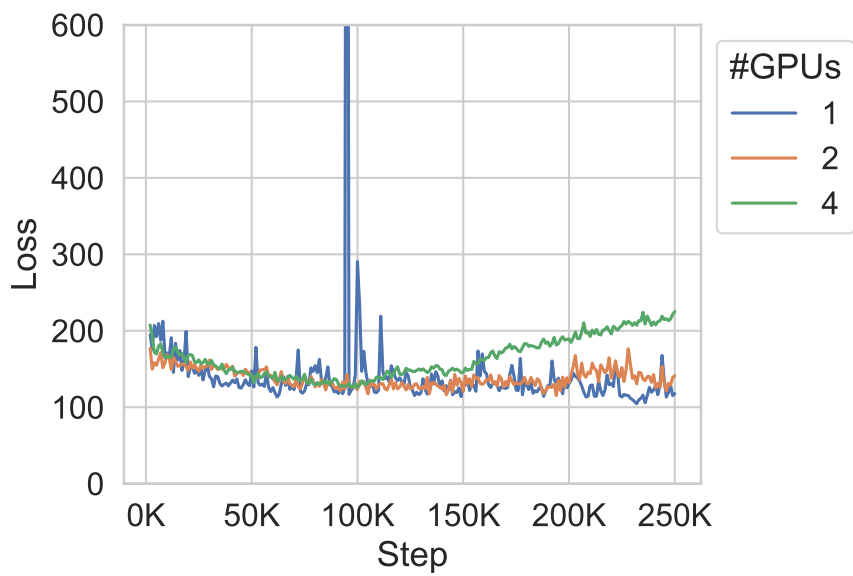


(b) Batch Size = 1024

Figure 6.4: Accuracy over time.



(a) Batch Size = 64



(b) Batch Size = 1024

Figure 6.5: Loss over time.

et al., 2022). A convolutional neural network takes as input a frame pair (frames are concatenated along the channel dimension) and it outputs the displacement components for all the pixels. We investigate online unsupervised learning, i.e. training is performed using a single frame pair at every step, and the pairs are sequentially extracted from a video source without shuffling. Unsupervised learning is driven by the brightness constancy assumption coupled with spatial regularization (see (Marullo et al., 2022) for details). As video source, we choose “1917”, a 2019 British war film directed and produced by Sam Mendes. The film lasts approximately 103 minutes (without credits) and appears as a single long continuous take, without artificial cuts. We report in Figure 6.6 the optical flow estimated by the sequential and pipelined network. We selected a learning rate $\mu = 10^{-5}$ for both the sequential and the 2-stage pipeline, while it is set to $\mu = 10^{-6}$ for the 4-stage pipeline to account for the assumption of slowly changing gradients (see section 6.2). Noticeably, the estimated flow from the pipelined network is qualitatively similar to that of the sequential network, with little degradation in the 4-stages case. Figure 6.9 shows how the training loss changes every 1-minute window in the movie. Whilst the gradients computed by the pipelined model are more subject to approximation errors with an increased number of stages, we remark that the learning curve follows the same patterns of the vanilla sequential model, yielding a valid learning process.

As expected by the theoretical analysis in Section 6.2 the pipeline gradients degrade and make learning more difficult due to parts of the movie where there are intense and rapid movements that interests many part of the images, weakening the assumption of similar inputs. Even so, the learning trend appears very similar and it is very close to sequential learning in the rest of the movie.

6.5 Discussion

In this chapter we discussed PARTIME, a Python software library designed for continual learning problems in which the data is streamed over time. PARTIME is built on a pipeline parallelism that speedups the computations of a neural network by a theoretical $\times D$, being D the number of devices. We focused on the case of Graphics Processing Units (GPUs), showing that our implementation scales coherently with the expectations, as experimented in an environment with up to 8 GPUs. PARTIME is easily inserted in a context where a researcher wants to process and learn from real-time online streams, but needs to use a feed forward network that is unable to run at real-time framerate on a single accelerator. As mentioned, a clear example of this is the streams generated by Virtual Environments, that can offer a great quantity of data at real-time speeds, but are often not closely followed by inference and learning speeds of neural networks. Future work will consider and improve the implementation for specific classes of neural models, such as those that are recurrent

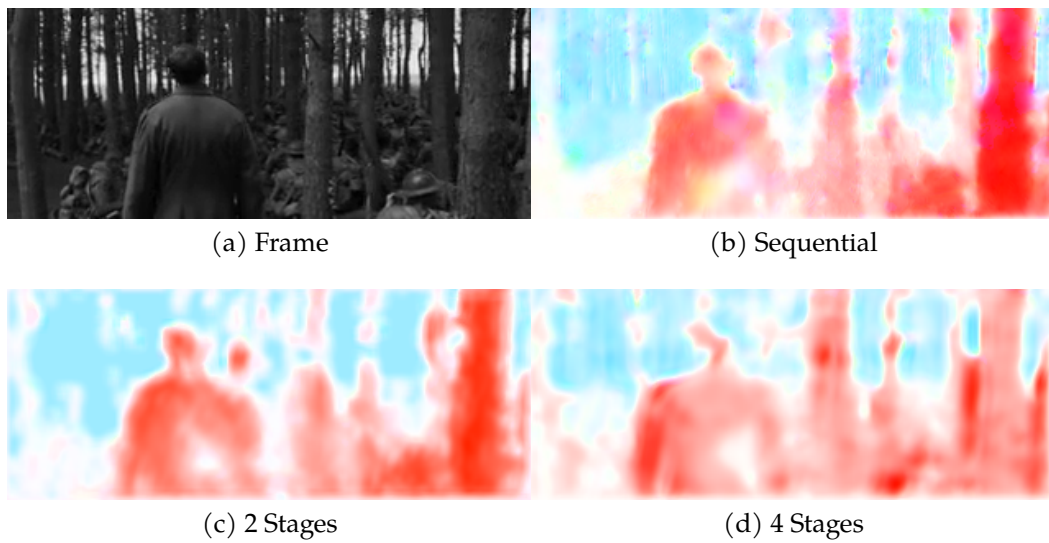


Figure 6.6: Captures of Optical Flow estimation on a given frame (Figure 6.6a) by the vanilla sequential network (Figure 6.6b), compared with the Optical Flow computed by the network handled by PARTIME (Figure 6.6c, Figure 6.6d).

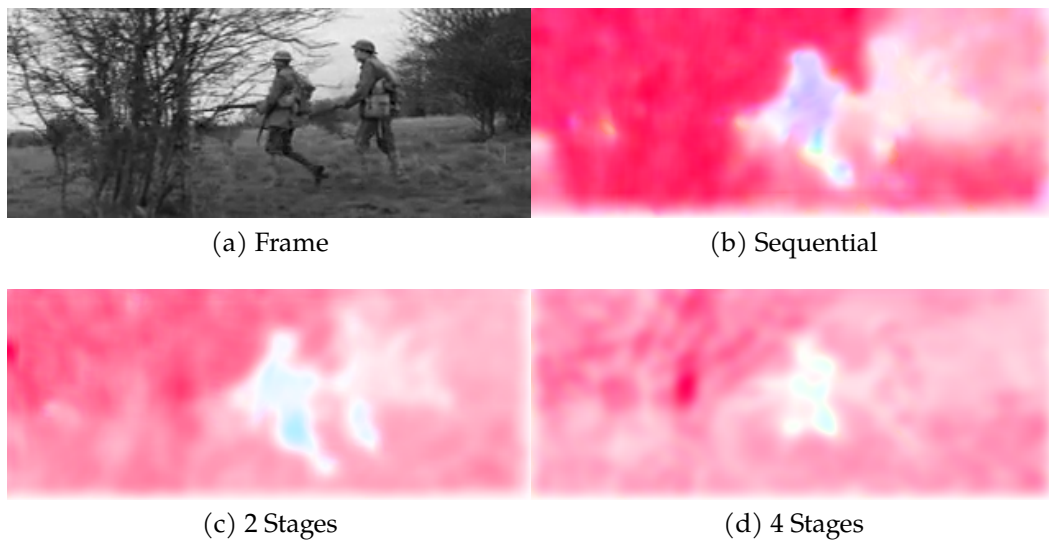


Figure 6.7: Captures of Optical Flow estimation on a given frame (Figure 6.7a) by the vanilla sequential network (Figure 6.7b), compared with the Optical Flow computed by the network handled by PARTIME (Figure 6.7c, Figure 6.7d).

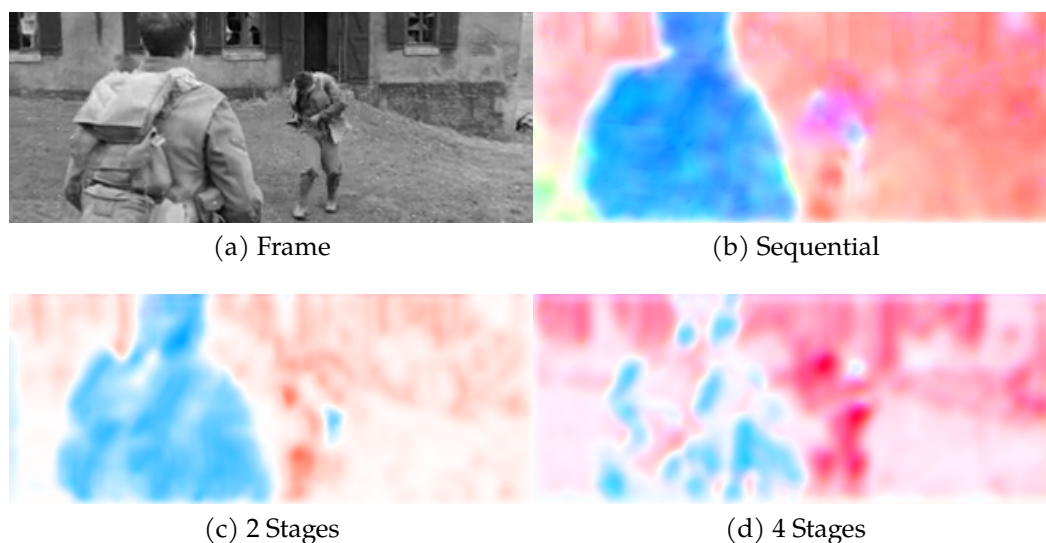


Figure 6.8: Captures of Optical Flow estimation on a given frame (Figure 6.8a) by the vanilla sequential network (Figure 6.8b), compared with the Optical Flow computed by the network handled by PARTIME (Figure 6.8c, Figure 6.8d).

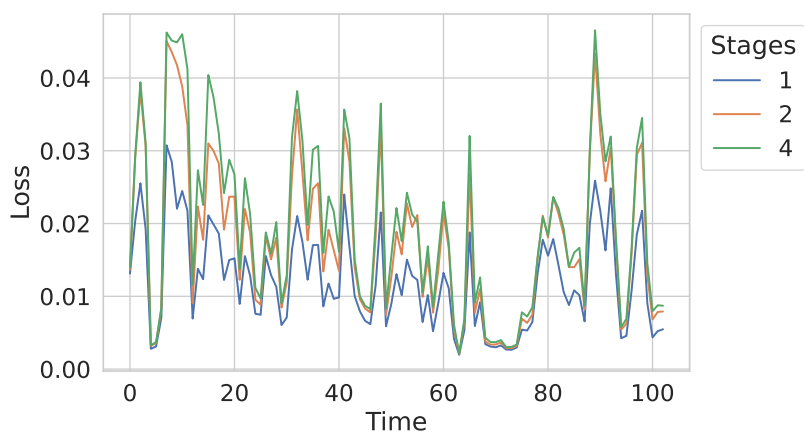


Figure 6.9: Continual online Optical Flow training loss over time (y-axes) over 1-minute windows on which the average loss is plotted (x-axis).

and neural networks for graphs, as well to local optimization approaches (Marra et al., 2020), as well as further experiments directly involving Virtual Environments and real-time Lifelong Learning.

Chapter 7

Conclusions

In this chapter we summarize the work presented throughout this thesis in the direction of novel engineering solutions for opening new possibilities and scenarios in research work, for general Computer Vision and in particular for Continual Learning, Adversarial Learning, Optical Flow Estimation and Real-Time Online Learning.

These contributions revolve around the idea of a Virtual Environment that is able to generate an annotated real-time photo-realistic stream that suits the needs of researchers that wish to relinquish the assumption of having a huge dataset on which to perform offline batch-mode learning and instead focuses on a paradigm of Learning with living agents that learn from online real-time multi-sensory stimuli as long as they live in an environment and without a neat distinction between training and testing phases, similarly to how humans learn in the real world.

Finally, we present some possible developments for the research works described in the thesis, providing some hopefully interesting insight through follow-up questions that might spark interest in new research.

7.1 Summary of Contributions

- In chapter 3 we introduced SAILenv, a 3D Virtual Environment, specifically designed for visual recognition tasks and implemented to be easy to use and extend by researchers interested in the field but without advanced expertise in the field of Computer Graphics. The framework comes with a library populated with 3D photo-realistic objects that can be readily plugged in new scenarios to produce scenes of interest for various Computer Vision related experiments. With a few lines of code the platform can be integrated with the most common Machine Learning frameworks and used to quickly prototype experiments or scenarios. Furthermore, the environment takes advantage of a low-level communication protocol that reduces at its minimum the synchronization overheads. At the time of development, SAILenv was the first 3D

Virtual Environment to provide real-time motion information of the objects in the scene. Finally, we proved the photo-realistic quality of the environment through evaluation with a state-of-the-art pixel-wise object detector, and the efficiency in terms of framerates and response times of the generated stream. In conclusion, we listed a few articles that used the platform to enable experiments that would have been harder without it.

- In chapter 4 we described a theoretical parametric framework for the description and generation of visual scenes, thought with the intent of allowing researchers to create reproducible benchmarks for experiments in the field of Continual Learning. We define a formalization of the description of the scene, considering parameters that describe the trajectory of objects, their moment of appearance and disappearance and various other properties. We also implement it into SAILenv, proving its ease of extension and its flexibility as a tool for enabling different fields of research. After exemplifying the use of the implemented framework with few lines of code, we show graphical examples of scenes generated with such tools.
- in chapter 5 we studied the idea of transferring 3D Adversarial Objects from a simpler differentiable renderer (Surrogate Renderer) to a much more complex and photo-realistic, but not differentiable, renderer (Target Renderer). We argue the possibilities opened by Adversarial Attacks on Virtual Environment, both as a malicious way of poisoning benchmarks, and as a way to evaluate and improve the robustness of neural models to Adversarial Attacks. We propose a saliency-based variation of Projected Gradient Descent that aims at carefully attacking pixels of the object that are known to be important in the classification of the image produced by the Target Renderer. Then, we implement the proposed attack method in PyTorch, transferring attacks from a popular differentiable render, PyTorch3D, to SAILenv, which takes advantage of the widely used industrial level renderer of the Unity Engine. We design a set of experiments to evaluate the effectiveness of the attack and of the transferral to the Target Renderer, measuring the drop in classification accuracy, considering the percentage of altered pixels as a measure of how “noticeable” the attack is by a human observer. The result are then discussed, showing how the attacks are generally effective even though the differences between Surrogate and Target renderer are clearly evident and impactful.
- In chapter 6 we introduce PARTIME, a Python library meant for wrapping neural feed-forward models to transparently enable multi-GPU pipeline parallelism, with the aim of significantly increase output framerates in wrapped networks to allow for Real-time Online Learning. We discuss the differences with state-of-the-art methods of Pipeline Parallelism, mostly designed to ac-

celerate batch-mode offline processing. We introduce an approximation of the Backpropagation algorithm that allows the library to work at real-time speeds even without batch-mode processing and study its implications in the quality of the learning, providing reasonable assumptions on the nature of the input data. We design a set of experiments meant to evaluate both the learning quality of the wrapped network, comparing it to the vanilla case, and similarly compare the relative speed-ups achieved by the multi-GPU parallelism. The results are compatible with the theoretical analysis and corroborate it, showing almost linear speed-ups in many tasks and a quality of learned parameters that is acceptable with respect to the significant improvement in training time, and also considering the use of networks specifically designed for batch-mode offline training.

7.2 Future work

The research work described in the thesis open up new questions and leave some open problems up to future research. We propose a few directions, giving some insights to the possible questions that could be answered.

Interactions and Reinforcement Learning support in SAILenv. While SAILenv is a powerful platform for visual recognition tasks, as of now it does not support interactions between agents and the environment or even other agents. This is a possibility that was already studied by AI2-THOR, but given its nature, it works as a discrete machine where the agent actions dictate the flow of time in the environment. Differently, the flow of time in SAILenv is detached by the actions of the agents, introducing further difficulties and important problems to be solved while enabling important questions. For example, when times are synchronized, multiple agents can act at the same time, which is an important yet unrealistic assumption in the world model. Allowing interactions in an asynchronous environment allows the researchers to study the effects on interactions between agents that experience the flow of time in an independent way.

Outdoor scenes in SAILenv. SAILenv principally focuses on indoor environments, which are by nature limited in space and presenting a limited range of possible objects, movements are almost always restricted between the walls with limited velocities, distances and scales are constrained allowing reduced variability. SAILenv would gain significant value by introducing outdoor environments, allowing new types of experiments with an increased library of photo-realistic objects. Many Computer Vision problems could then be reframed in the Virtual Environment, taking advantage of the generation of real-time data on variable scenarios with extensible

and configurable behaviors.

Renderer Imitation to improve Adversarial Attacks. One of the focal points that limit the effectiveness of transferred objects is certainly the strong differences between Surrogate Renderers and Adversarial Renderers. Even considering the implementation of a PBR differentiable renderer that easily integrates with widely used Machine Learning frameworks, which is by itself a promising direction, an issue would still remain: usually, Target Renderers are closed source and there is no guarantee that the renderer will be able to properly render images close enough to produce a 100% effective attack. Therefore, taking inspiration from Neural Renderers, it is an interesting approach to create learnable PBR renderer, that taking as input the standard textures used by PBR renderers, learn from images rendered by a Target Renderer how to replicate the general appearance and particular details of the Target Renderer. Such a renderer can then be readily used to generate what we hypothesize will be easily and effectively transferrable 3D Adversarial Objects.

Extension of wrappable network architectures in PARTIME. PARTIME allows feed-forward architectures to be wrapped and parallelized. While it is certainly true that many problems are solved through the use of feed-forward networks, there is a ever-growing quantity of research that takes advantage of recurrent networks or transformers. Therefore, an important step forward for enabling multi-GPU Pipeline Parallelism for a varied array of tasks is designing an extension of PARTIME for models that do not conform to the simple feed-forward structure. This, of course, implies the extension of the computational model of PARTIME, as well as the theoretical analysis of the gradient that needs to be redesigned to account for possible cycles.

Better approximate gradients in PARTIME. As we have seen, while the approximation of the gradients used in PARTIME allows for linear speed-up in computations, it still degrades the gradients that reach the parts of the network closer to the input. An important step in the direction of increasing the usefulness of this tool is to study new approximations of the gradients. An important step in this direction was made by the work on PipeDream-2BW, which enables fast computations while limiting the gradient degradation and the memory footprint, if only for the batch-mode offline case. We hypothesize that this idea can be extended to the real-time online case, still maintaining high speed-ups while achieving minor gradient degradation and significant learning quality. Furthermore, these new approximation should consider the case of different network architectures in their design.

Appendix A

Publications

Journal papers

1. Marco Di Benedetto, Fabio Carrara, **Enrico Meloni**, Giuseppe Amato, Fabrizio Falchi, Claudio Gennaro, “Learning Accurate Personal Protective Equipment Detection from Virtual Worlds”, *Multimedia Tools and Applications* 80, pages:23241—23253, 2021. **Candidate’s contribution:** Development of software tools, review of literature, design of experiments, analysis of experimental results, generation of synthetic dataset, manual annotation of real-world images dataset.

Peer reviewed conference papers

1. Marco Di Benedetto, **Enrico Meloni**, Giuseppe Amato, Fabrizio Falchi, Claudio Gennaro, “Learning Safety Equipment Detection using Virtual Worlds”, *International Conference on Content-Based Multimedia Indexing (CBMI)*, 2019. **Candidate’s contributions:** Development of software tools, review of literature, design of experiments, analysis of experimental results, generation of synthetic dataset, manual annotation of real-world images dataset.
2. **Enrico Meloni**, Luca Pasqualini, Matteo Tiezzi, Marco Gori, Stefano Melacci, “SAIEnv: Learning in Virtual Visual Environments Made Simple”, *International Conference on Pattern Recognition (ICPR)*, 2020. **Candidate’s contributions:** Development of software tools, review of literature, design of experiments, analysis of experimental results.
3. **Enrico Meloni**, Matteo Tiezzi, Luca Pasqualini, Marco Gori, Stefano Melacci, “Messing Up 3D Virtual Environments: Transferable Adversarial 3D Objects”, *International Conference on Machine Learning and Applications (ICMLA)*, 2021. **Candidate’s contributions:** Development of software tools, review of literature, design of experiments, analysis of experimental results.

4. **Enrico Meloni**, Lapo Faggi, Simone Marullo, Alessandro Betti, Matteo Tiezzi, Marco Gori, Stefano Melacci, “PARTIME: Scalable and Parallel Processing Over Time with Deep Neural Networks”, *International Conference on Machine Learning and Applications (ICMLA)*, 2022. **Candidate’s contributions:** Development of software tools, review of literature, design of experiments, analysis of experimental results.
5. Matteo Tiezzi, Simone Marullo, Lapo Faggi, **Enrico Meloni**, Alessandro Betti, Stefano Melacci, “Stochastic Coherence Over Attention Trajectory For Continuous Learning In Video Streams”, *International Joint Conference on Artificial Intelligence (IJCAI-ECAI)*, 2022. **Candidate’s contributions:** Generation of synthetic visual streams, design of algorithms, discussions, review of paper.
6. Simone Marullo, Matteo Tiezzi, Alessandro Betti, Lapo Faggi **Enrico Meloni**, Stefano Melacci, ‘Continual Unsupervised Learning for Optical Flow Estimation’, *Conference on Lifelong Learning Agents (CoLLAs)*, 2022. **Candidate’s contributions:** Generation of synthetic visual streams, discussions, review of paper.
7. Matteo Tiezzi, Simone Marullo, Alessandro Betti, **Enrico Meloni**, Lapo Faggi, Marco Gori, Stefano Melacci, “Foveated Neural Computation”, *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD)*, 2022. **Candidate’s contributions:** Implementation of algorithm, discussion, review of paper.
8. Alessandro Betti, Lapo Faggi, Marco Gori, Matteo Tiezzi, Simone Marullo, **Enrico Meloni**, Stefano Melacci, “Continual Learning through Hamilton Equations”, *Conference on Lifelong Learning Agents (CoLLAs)*, 2022. **Candidate’s contributions:** Discussions, review of paper.

Workshop papers

1. **Enrico Meloni**, Alessandro Betti, Lapo Faggi, Simone Marullo, Matteo Tiezzi, Stefano Melacci, “Evaluating Continual Learning Algorithms by Generating 3D Virtual Environments”, *International Workshop on Continual Semi-Supervised Learning*, pages:62–74, 2022. **Candidate’s contributions:** Design of theoretical framework, development of software tools, review of literature.

Papers under review

None.

Other

1. Andrea Zugarini, **Enrico Meloni**, Alessandro Betti, Andrea Panizza, Marco Corneli, Marco Gori, "An Optimal Control Approach to Learning in SIDARTHE Epidemic model", *arXiv pre-print*, 2020. **Candidate's contributions:** Design of algorithms, design of experiments, development of software tools, review of literature, analysis of experimental results.

Bibliography

- Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.
- Abraham, W. C. and Robins, A. (2005). Memory retention—the synaptic stability versus plasticity dilemma. *Trends in neurosciences*, 28(2):73–78.
- Akhtar, N. and Mian, A. S. (2018). Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430.
- Alcorn, M. A., Li, Q., Gong, Z., Wang, C., Mai, L., Ku, W.-S., and Nguyen, A. (2019). Strike (with) a pose: Neural networks are easily fooled by strange poses of familiar objects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4845–4854.
- Aljundi, R., Kelchtermans, K., and Tuytelaars, T. (2019). Task-free continual learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11254–11263.
- Athalye, A., Engstrom, L., Ilyas, A., and Kwok, K. (2018). Synthesizing robust adversarial examples. In *International Conference on Machine Learning*, pages 284–293. PMLR.
- Azinovic, D., Li, T.-M., Kaplanyan, A., and Nießner, M. (2019). Inverse path tracing for joint material and lighting estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2447–2456.
- Barreno, M., Nelson, B., Joseph, A. D., and Tygar, J. D. (2010). The security of machine learning. *Machine Learning*, 81(2):121–148.
- Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., et al. (2016). Deepmind lab. *arXiv:1612.03801*.
- Betti, A. and Gori, M. (2019). Backprop diffusion is biologically plausible.

- Betti, A., Gori, M., Marullo, S., and Melacci, S. (2020a). Developing constrained neural units over time. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.
- Betti, A., Gori, M., and Melacci, S. (2020b). Cognitive action laws: The case of visual features. *IEEE Transactions on Neural Networks and Learning Systems*, 31(3):938–949.
- Betti, A., Gori, M., and Melacci, S. (2020c). Learning visual features under motion invariance. *Neural Networks*, 126:275–299.
- Betti, A., Gori, M., and Melacci, S. (2022). *Deep Learning to See - Towards New Foundations of Computer Vision*. Springer.
- Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., and Roli, F. (2013). Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 387–402. Springer.
- Biggio, B. and Roli, F. (2018). Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331.
- Brady, M. L., Raghavan, R., and Slawny, J. (1989). Back propagation fails to separate where perceptrons succeed. *IEEE Transactions on Circuits and Systems*, 36(5):665–674.
- Bremner, A. J., Lewkowicz, D. J., and Spence, C. (2012). *Multisensory development*. Oxford University Press.
- Brodeur, S., Perez, E., Anand, A., Golemo, F., Celotti, L., Strub, F., Rouat, J., Larochelle, H., and Courville, A. (2017). Home: A household multimodal environment. *arXiv preprint arXiv:1711.11017*.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.
- Brox, T., Bruhn, A., Papenberger, N., and Weickert, J. (2004). High accuracy optical flow estimation based on a theory for warping. In *European conference on computer vision*, pages 25–36. Springer.
- Calvert, G., Spence, C., Stein, B. E., et al. (2004). *The handbook of multisensory processes*. MIT press.
- Cangelosi, A. and Schlesinger, M. (2018). From babies to robots: the contribution of developmental robotics to developmental psychology. *Child Development Perspectives*, 12(3):183–188.

- Carlini, N., Mishra, P., Vaidya, T., Zhang, Y., Sherr, M., Shields, C., Wagner, D., and Zhou, W. (2016). Hidden voice commands. In *25th USENIX security symposium (USENIX security 16)*, pages 513–530.
- Chakraborty, A., Alam, M., Dey, V., Chattopadhyay, A., and Mukhopadhyay, D. (2018). Adversarial attacks and defences: A survey. *arXiv preprint arXiv:1810.00069*.
- Chang, A., Dai, A., Funkhouser, T., Halber, M., Niessner, M., Savva, M., Song, S., Zeng, A., and Zhang, Y. (2017). Matterport3d: Learning from rgb-d data in indoor environments. *arXiv preprint arXiv:1709.06158*.
- Chaplot, D. S., Sathyendra, K. M., Pasumarthi, R. K., Rajagopal, D., and Salakhutdinov, R. (2018). Gated-attention architectures for task-oriented language grounding. In *Thirty-Second AAAI Conference on Artificial Intelligence*.
- Chaudhry, A., Rohrbach, M., Elhoseiny, M., Ajanthan, T., Dokania, P. K., Torr, P. H., and Ranzato, M. (2019). Continual learning with tiny episodic memories.
- Chellapilla, K., Puri, S., and Simard, P. (2006). High performance convolutional neural networks for document processing. In *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft.
- Chen, B., Carvalho, W., Baracaldo, N., Ludwig, H., Edwards, B., Lee, T., Molloy, I., and Srivastava, B. (2018). Detecting backdoor attacks on deep neural networks by activation clustering. *arXiv preprint arXiv:1811.03728*.
- Chen, W., Ling, H., Gao, J., Smith, E., Lehtinen, J., Jacobson, A., and Fidler, S. (2019). Learning to predict 3d objects with an interpolation-based differentiable renderer. *Advances in Neural Information Processing Systems*, 32.
- Chen, Z. and Liu, B. (2018). Lifelong machine learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 12(3):1–207.
- Cheng, J., Grossman, M., and McKercher, T. (2014). *Professional CUDA c programming*. John Wiley & Sons.
- Cireřan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220.
- De Lange, M., Aljundi, R., Masana, M., Parisot, S., Jia, X., Leonardis, A., Slabaugh, G., and Tuytelaars, T. (2021). A continual learning survey: Defying forgetting in classification tasks. *IEEE transactions on pattern analysis and machine intelligence*, 44(7):3366–3385.

- Deitke, M., Han, W., Herrasti, A., Kembhavi, A., Kolve, E., et al. (2020). Robothor: An open simulation-to-real embodied ai platform. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3164–3174.
- Delange, M., Aljundi, R., Masana, M., Parisot, S., Jia, X., Leonardis, A., Slabaugh, G., and Tuytelaars, T. (2021). A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1.
- Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee.
- Di Benedetto, M., Carrara, F., Meloni, E., Amato, G., Falchi, F., and Gennaro, C. (2021). Learning accurate personal protective equipment detection from virtual worlds. *Multimedia Tools and Applications*, 80(15):23241–23253.
- Di Benedetto, M., Meloni, E., Amato, G., Falchi, F., and Gennaro, C. (2019). Learning safety equipment detection using virtual worlds. In *2019 International Conference on Content-Based Multimedia Indexing (CBMI)*, pages 1–6. IEEE.
- Ditzler, G., Roveri, M., Alippi, C., and Polikar, R. (2015). Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine*, 10(4):12–25.
- Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. (2017). CARLA: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR.
- Fabbri, M., Lanzi, F., Calderara, S., Palazzi, A., Vezzani, R., and Cucchiara, R. (2018). Learning to detect and track visible and occluded body joints in a virtual world. In *European Conference on Computer Vision (ECCV)*.
- Fanello, S., Ciliberto, C., Santoro, M., Natale, L., Metta, G., Rosasco, L., and Odone, F. (2013). icub world: Friendly robots help building good vision data-sets. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 700–705.
- Farnebäck, G. (2003). Two-frame motion estimation based on polynomial expansion. In *Scandinavian conference on Image analysis*, pages 363–370. Springer.
- Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A. A., Pritzel, A., and Wierstra, D. (2017). Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*.
- Fradkov, A. L. (2020). Early history of machine learning. *IFAC-PapersOnLine*, 53(2):1385–1390.

- French, R. M. (1999). Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135.
- Gallardo, J., Hayes, T. L., and Kanan, C. (2021). Self-supervised training enhances online continual learning. *CoRR*.
- Gan, C., Schwartz, J., Alter, S., Schrimpf, M., et al. (2020). Threedworld: A platform for interactive multi-modal physical simulation. *arXiv:2007.04954*.
- Gao, X., Gong, R., Shu, T., Xie, X., Wang, S., and Zhu, S.-C. (2019). Vrkitchen: an interactive 3d virtual environment for task-oriented learning. *arXiv:1903.05757*.
- Genova, K., Cole, F., Maschinot, A., Sarna, A., Vlastic, D., and Freeman, W. T. (2018). Unsupervised training for 3d morphable model regression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8377–8386.
- Gilmer, J., Adams, R. P., Goodfellow, I., Andersen, D., and Dahl, G. E. (2018). Motivating the rules of the game for adversarial example research. *arXiv preprint arXiv:1807.06732*.
- Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014). Explaining and harnessing adversarial examples. *arXiv:1412.6572*.
- Gopnik, A., Meltzoff, A. N., and Kuhl, P. K. (1999). *The scientist in the crib: Minds, brains, and how children learn*. William Morrow & Co.
- Gordon, D., Kembhavi, A., Rastegari, M., Redmon, J., Fox, D., and Farhadi, A. (2018). Iqa: Visual question answering in interactive environments. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4089–4098.
- Gori, M. and Tesi, A. (1992). On the problem of local minima in backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(1):76–86.
- Grigorescu, S., Trasnea, B., Cocias, T., and Macesanu, G. (2020). A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386.
- Grossberg, S. (1982). How does a brain build a cognitive code? In *Studies of mind and brain*, pages 1–52. Springer.
- Grossberg, S. (2013). Adaptive resonance theory: How a brain learns to consciously attend, learn, and recognize a changing world. *Neural networks*, 37:1–47.
- Guan, L., Yin, W., Li, D., and Lu, X. (2019). Xpipe: Efficient pipeline model parallelism for multi-gpu dnn training.

- Guo, Y., Liu, Y., Oerlemans, A., Lao, S., Wu, S., and Lew, M. S. (2016). Deep learning for visual understanding: A review. *Neurocomputing*, 187:27–48.
- Gupta, S., Davidson, J., Levine, S., Sukthankar, R., and Malik, J. (2017). Cognitive mapping and planning for visual navigation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2616–2625.
- Haas, J. K. (2014). A history of the unity game engine.
- Handa, A., Pătrăucean, V., Stent, S., and Cipolla, R. (2016). Scenenet: An annotated model generator for indoor scene understanding. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5737–5743. IEEE.
- Hassabis, D., Kumaran, D., Summerfield, C., and Botvinick, M. (2017). Neuroscience-inspired artificial intelligence. *Neuron*, 95(2):245–258.
- He, K., Gkioxari, G., Dollár, P., and Girshick, R. (2017). Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969.
- He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.
- Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. (2018). Gpipe: Efficient training of giant neural networks using pipeline parallelism.
- Hui, T.-W., Tang, X., and Loy, C. C. (2018). LiteFlowNet: A lightweight convolutional neural network for optical flow estimation. In *IEEE Conference on Computer Vision and Pattern Recognition*.
- Isele, D. and Cosgun, A. (2018). Selective experience replay for lifelong learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.
- Jatavallabhula, K. M., Smith, E., Lafleche, J.-F., Tsang, C. F., Rozantsev, A., Chen, W., Xiang, T., Lebedev, R., and Fidler, S. (2019). Kaolin: A pytorch library for accelerating 3d deep learning research. *arXiv preprint arXiv:1911.05063*.
- Johnson-Roberson, M., Barto, C., Mehta, R., Sridhar, S. N., Rosaen, K., and Vasudevan, R. (2016). Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks? *arXiv preprint arXiv:1610.01983*.
- Kato, H. and Harada, T. (2019). Learning view priors for single-view 3d reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9778–9787.

- Kato, H., Ushiku, Y., and Harada, T. (2018). Neural 3d mesh renderer. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3907–3916.
- Kemker, R., McClure, M., Abitino, A., Hayes, T., and Kanan, C. (2018). Measuring catastrophic forgetting in neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.
- Kim, C., Lee, H., Jeong, M., Baek, W., Yoon, B., Kim, I., Lim, S., and Kim, S. (2020). torchpipe: On-the-fly pipeline parallelism for training giant models. *arXiv preprint arXiv:2004.09910*.
- Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.
- Kolve, E., Mottaghi, R., Han, W., VanderBilt, E., Weihs, L., Herrasti, A., Gordon, D., Zhu, Y., Gupta, A., and Farhadi, A. (2017). Ai2-thor: An interactive 3d environment for visual ai. *arXiv:1712.05474*.
- Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90.
- Krueger, K. A. and Dayan, P. (2009). Flexible shaping: How learning in small steps helps. *Cognition*, 110(3):380–394.
- Kumar, A. and Mehta, S. (2017). A survey on resilient machine learning. *arXiv preprint arXiv:1707.03184*.
- Kurakin, A., Goodfellow, I., and Bengio, S. (2016). Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*.
- LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.
- LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., and Jackel, L. (1989a). Handwritten digit recognition with a back-propagation network. In Touretzky, D., editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann.
- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989b). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551.

- Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.
- Lerer, A., Gross, S., and Fergus, R. (2016). Learning physical intuition of block towers by example. *arXiv preprint arXiv:1603.01312*.
- Lewkowicz, D. J. (2014). Early experience and multisensory perceptual narrowing. *Developmental psychobiology*, 56(2):292–315.
- Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. (2020). Pytorch distributed: Experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12):3005–3018.
- Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.
- Liu, A., Huang, T., Liu, X., Xu, Y., Ma, Y., Chen, X., Maybank, S. J., and Tao, D. (2020). Spatiotemporal attacks for embodied agents. In *European Conference on Computer Vision*, pages 122–138. Springer.
- Liu, H.-T. D., Tao, M., Li, C.-L., Nowrouzezahrai, D., and Jacobson, A. (2018). Beyond pixel norm-balls: Parametric adversaries using an analytically differentiable renderer. In *International Conference on Learning Representations*.
- Liu, S., Li, T., Chen, W., and Li, H. (2019). Soft rasterizer: A differentiable renderer for image-based 3d reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7708–7717.
- Liu, V. and Curran, J. R. (2006). Web text corpus for natural language processing. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, pages 233–240.
- Lomonaco, V., Desai, K., Culurciello, E., and Maltoni, D. (2020). Continual reinforcement learning in 3d non-stationary environments. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 248–249.
- Lomonaco, V. and Maltoni, D. (2017). Core50: a new dataset and benchmark for continuous object recognition. In *Conference on Robot Learning*, pages 17–26. PMLR.
- Loper, M. M. and Black, M. J. (2014). Opendr: An approximate differentiable renderer. In *European Conference on Computer Vision*, pages 154–169. Springer.

- Lu, J., Sibai, H., Fabry, E., and Forsyth, D. (2017). No need to worry about adversarial examples in object detection in autonomous vehicles. *arXiv:1707.03501*.
- Luo, Y., Boix, X., Roig, G., Poggio, T., and Zhao, Q. (2015). Foveation-based mechanisms alleviate adversarial examples. *arXiv preprint arXiv:1511.06292*.
- Madaan, D., Yoon, J., Li, Y., Liu, Y., and Hwang, S. J. (2022). Representational continuity for unsupervised continual learning. In *International Conference on Learning Representations*.
- Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. (2017). Towards deep learning models resistant to adversarial attacks. *arXiv:1706.06083*.
- Maggini, M., Marra, G., Melacci, S., and Zugarini, A. (2020). Learning in text streams: Discovery and disambiguation of entity and relation instances. *IEEE Transactions on Neural Networks and Learning Systems*, 31(11):4475–4486.
- Maggini, M., Melacci, S., and Sarti, L. (2007). Representation of facial features by catmull-rom splines. In *International Conference on Computer Analysis of Images and Patterns*, pages 408–415. Springer.
- Mallya, A. and Lazebnik, S. (2018). Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 7765–7773.
- Maltoni, D. and Lomonaco, V. (2019). Continuous learning in single-incremental-task scenarios. *Neural Networks*, 116:56–73.
- Marra, G., Tiezzi, M., Melacci, S., Betti, A., Maggini, M., and Gori, M. (2020). Local propagation in constraint-based neural networks. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.
- Marullo, S., Tiezzi, M., Betti, A., Faggi, L., Meloni, E., and Melacci, S. (2022). Continual unsupervised learning for optical flow estimation with deep networks. In *Proceedings of the Conference on Lifelong Learning Agents (CoLLAs) 2022*. Available at <https://sailab.diism.unisi.it/continual-unsupervised-learning-for-optical-flow-estimation/> while waiting for publication.
- Mayer, N., Ilg, E., Fischer, P., Hazirbas, C., Cremers, D., Dosovitskiy, A., and Brox, T. (2018). What makes good synthetic training data for learning disparity and optical flow estimation? *International Journal of Computer Vision*, 126(9):942–960.
- Mayer, N., Ilg, E., Hausser, P., Fischer, P., Cremers, D., Dosovitskiy, A., and Brox, T. (2016). A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4040–4048.

- McClelland, J. L., McNaughton, B. L., and O'Reilly, R. C. (1995). Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review*, 102(3):419.
- McCloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier.
- McCormac, J., Handa, A., Leutenegger, S., and Davison, A. J. (2017). Scenenet rgb-d: Can 5m synthetic images beat generic imagenet pre-training on indoor segmentation? In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.
- Meloni, E. (2019). Using virtual worlds to train an object detector for personal protection equipment.
- Meloni, E., Betti, A., Faggi, L., Marullo, S., Tiezzi, M., and Melacci, S. (2022a). Evaluating continual learning algorithms by generating 3d virtual environments. In *International Workshop on Continual Semi-Supervised Learning*, pages 62–74. Springer.
- Meloni, E., Faggi, L., Marullo, S., Betti, A., Tiezzi, M., Gori, M., and Melacci, S. (2022b). Partime: Scalable and parallel processing over time with deep neural networks. *Accepted at 2022 21st IEEE International Conference on Machine Learning and Applications (ICMLA)*.
- Meloni, E., Pasqualini, L., Tiezzi, M., Gori, M., and Melacci, S. (2021a). Sailenv: Learning in virtual visual environments made simple. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 8906–8913. IEEE.
- Meloni, E., Tiezzi, M., Pasqualini, L., Gori, M., and Melacci, S. (2021b). Messing up 3d virtual environments: Transferable adversarial 3d objects. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1–8. IEEE.
- Mermillod, M., Bugaiska, A., and Bonin, P. (2013). The stability-plasticity dilemma: Investigating the continuum from catastrophic forgetting to age-limited learning effects.
- Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., and Ng, R. (2020). Nerf: Representing scenes as neural radiance fields for view synthesis. In *European Conference on Computer Vision*, pages 405–421. Springer.
- Murray, M. M., Lewkowicz, D. J., Amedi, A., and Wallace, M. T. (2016). Multi-sensory processes: a balancing act across the lifespan. *Trends in Neurosciences*, 39(8):567–579.

- Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. (2019). Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15.
- Narayanan, D., Phanishayee, A., Shi, K., Chen, X., and Zaharia, M. (2021). Memory-efficient pipeline-parallel dnn training. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 7937–7947. PMLR.
- Nguyen, C. V., Li, Y., Bui, T. D., and Turner, R. E. (2017). Variational continual learning. *arXiv preprint arXiv:1710.10628*.
- Nimier-David, M., Vicini, D., Zeltner, T., and Jakob, W. (2019). Mitsuba 2: A re-targetable forward and inverse renderer. *ACM Transactions on Graphics (TOG)*, 38(6):1–17.
- Parisi, G. I., Kemker, R., Part, J. L., Kanan, C., and Wermter, S. (2019). Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71.
- Pasquale, G., Ciliberto, C., Odone, F., Rosasco, L., and Natale, L. (2015). Teaching icub to recognize objects using deep convolutional neural networks. In *Machine Learning for Interactive Systems*, pages 21–25. PMLR.
- Pasquale, G., Ciliberto, C., Rosasco, L., and Natale, L. (2016). Object identification from few examples by improving the invariance of a deep convolutional neural network. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4904–4911. IEEE.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.
- Petrowski, A., Dreyfus, G., and Girault, C. (1993). Performance analysis of a pipelined backpropagation parallel algorithm. *IEEE Transactions on Neural Networks*, 4(6):970–981.
- Puig, X., Ra, K., Boben, M., Li, J., Wang, T., Fidler, S., and Torralba, A. (2018). Virtualhome: Simulating household activities via programs. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 8494–8502.

- Quiter, C. and Ernst, M. (2018). deepdrive/deepdrive: 2.0. URL: [https://doi.org/10.5281/zenodo, 1248998](https://doi.org/10.5281/zenodo.1248998).
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.
- Rao, R. (2000). Reinforcement learning: An introduction; rs sutton, ag barto (eds.); mit press, cambridge, ma, 1998, 380 pages, isbn 0-262-19398-1.
- Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. (2020). Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD 2020*, pages 3505–3506.
- Ravi, N., Reizenstein, J., Novotny, D., Gordon, T., Lo, W.-Y., Johnson, J., and Gkioxari, G. (2020). Accelerating 3d deep learning with pytorch3d. *arXiv:2007.08501*.
- Rebuffi, S.-A., Kolesnikov, A., Sperl, G., and Lampert, C. H. (2017). icarl: Incremental classifier and representation learning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5533–5542.
- Rematas, K. and Ferrari, V. (2020). Neural voxel renderer: Learning an accurate and controllable rendering tool. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5417–5427.
- Rhodin, H., Robertini, N., Richardt, C., Seidel, H.-P., and Theobalt, C. (2015). A versatile scene model with differentiable visibility applied to generative pose estimation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 765–773.
- Rolnick, D., Ahuja, A., Schwarz, J., Lillicrap, T., and Wayne, G. (2019). Experience replay for continual learning. *Advances in Neural Information Processing Systems*, 32.
- Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.
- Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet

- Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252.
- Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. (2016). Progressive neural networks. *arXiv preprint arXiv:1606.04671*.
- Savva, M., Kadian, A., Maksymets, O., et al. (2019). Habitat: A platform for embodied ai research. In *IEEE/CVF International Conference on Computer Vision*, pages 9339–9347.
- Serra, J., Suris, D., Miron, M., and Karatzoglou, A. (2018). Overcoming catastrophic forgetting with hard attention to the task. In *International Conference on Machine Learning*, pages 4548–4557. PMLR.
- Shin, H., Lee, J. K., Kim, J., and Kim, J. (2017). Continual learning with deep generative replay. *Advances in neural information processing systems*, 30.
- Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. (2019). Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.
- Shridhar, M., Thomason, J., Gordon, D., Bisk, Y., Han, W., Mottaghi, R., Zettlemoyer, L., and Fox, D. (2020). Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10740–10749.
- Shu, T., Bhandwaldar, A., Gan, C., Smith, K., Liu, S., Gutfreund, D., Spelke, E., Tenenbaum, J., and Ullman, T. (2021). Agent: A benchmark for core psychological reasoning. In *International Conference on Machine Learning*, pages 9614–9625. PMLR.
- Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
- Song, S., Yu, F., Zeng, A., Chang, A. X., Savva, M., and Funkhouser, T. (2017). Semantic scene completion from a single depth image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1746–1754.
- Steinkraus, D., Buck, I., and Simard, P. (2005). Using gpus for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pages 1115–1120. IEEE.
- Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2013). Intriguing properties of neural networks. *arXiv:1312.6199*.

- Tani, J. (2016). *Exploring robotic minds: actions, symbols, and consciousness as self-organizing dynamic phenomena*. Oxford University Press.
- Thrun, S. and Mitchell, T. M. (1995). Lifelong robot learning. *Robotics and autonomous systems*, 15(1-2):25–46.
- Tiezzi, M., Marra, G., Melacci, S., and Maggini, M. (2022a). Deep constraint-based propagation in graph neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(2):727–739.
- Tiezzi, M., Marullo, S., Faggi, L., Meloni, E., Betti, A., and Melacci, S. (2022b). Stochastic coherence over attention trajectory for continuous learning in video streams.
- Toheed, A., Yousaf, M. H., Rabnawaz, and Javed, A. (2022). Physical adversarial attack scheme on object detectors using 3d adversarial object. In *2022 2nd International Conference on Digital Futures and Transformative Technologies (ICoDT2)*, pages 1–4.
- Van de Ven, G. M. and Tolias, A. S. (2019). Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.
- Wah, C., Branson, S., Welinder, P., Perona, P., and Belongie, S. (2011). The Caltech-UCSD Birds-200-2011 Dataset. Technical Report CNS-TR-2011-001, California Institute of Technology.
- Wang, X., Xiong, W., Wang, H., and Yang Wang, W. (2018). Look before you leap: Bridging model-free and model-based reinforcement learning for planned-ahead vision-and-language navigation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 37–53.
- Weihs, L., Salvador, J., Kotar, K., Jain, U., Zeng, K.-H., Mottaghi, R., and Kembhavi, A. (2020). Allenact: A framework for embodied ai research. *arXiv:2008.12760*.
- Xia, F., Shen, W. B., Li, C., Kasimbeg, P., Tchapmi, M. E., Toshev, A., Martín-Martín, R., and Savarese, S. (2020). Interactive gibbon benchmark: A benchmark for interactive navigation in cluttered environments. *IEEE Robotics and Automation Letters*, 5(2):713–720.
- Xia, F., Zamir, A. R., He, Z., Sax, A., Malik, J., and Savarese, S. (2018). Gibson env: Real-world perception for embodied agents. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9068–9079.

- Xiang, F., Qin, Y., Mo, K., et al. (2020). Sapien: A simulated part-based interactive environment. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11097–11107.
- Xiao, C., Yang, D., Li, B., Deng, J., and Liu, M. (2019). Meshadv: Adversarial meshes for visual recognition. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6898–6907.
- Xu, J. and Zhu, Z. (2018). Reinforced continual learning. *Advances in Neural Information Processing Systems*, 31.
- Yadan, O., Adams, K., Taigman, Y., and Ranzato, M. (2013). Multi-gpu training of convnets.
- Yan, C., Misra, D., Bennett, A., Walsman, A., Bisk, Y., and Artzi, Y. (2018). Chalet: Cornell house agent learning environment. *arXiv:1801.07357*.
- Yan, X., Yang, J., Yumer, E., Guo, Y., and Lee, H. (2016). Perspective transformer nets: Learning single-view 3d object reconstruction without 3d supervision. *Advances in neural information processing systems*, 29.
- Yao, P., So, A., Chen, T., and Ji, H. (2020). On multiview robustness of 3d adversarial attacks. In *Practice and Experience in Advanced Research Computing*, pages 372–378.
- Zeng, X., Liu, C., Wang, Y.-S., Qiu, W., Xie, L., Tai, Y.-W., Tang, C.-K., and Yuille, A. L. (2019). Adversarial attacks beyond the image space. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4302–4311.
- Zenke, F., Poole, B., and Ganguli, S. (2017). Continual learning through synaptic intelligence. In *International Conference on Machine Learning*, pages 3987–3995. PMLR.
- Zhang, G., Yan, C., Ji, X., Zhang, T., Zhang, T., and Xu, W. (2017). Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 103–117.
- Zhu, Y., Mottaghi, R., Kolve, E., Lim, J. J., Gupta, A., Fei-Fei, L., and Farhadi, A. (2017). Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3357–3364. IEEE.