Universidad de Las Palmas

de Gran Canaria

Doctoral Thesis

Contributions to a Methodology for the Building of Modular Neural Networks

David Castillo Bolado

Doctorado en Tecnologías de Telecomunicación e Ingeniería Computacional

Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería

Las Palmas de Gran Canaria

October 20, 2022

UNIVERSIDAD DE LAS PALMAS

DE GRAN CANARIA

DOCTORAL THESIS

Contributions to a Methodology for the Building of Modular Neural Networks

Author:

David Castillo Bolado

Supervisor:

Dr. Cayetano Guerra Artal

Doctorado en Tecnologías de Telecomunicación e Ingeniería Computacional

Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería

Las Palmas de Gran Canaria

October 20, 2022

“*Our intelligence is what makes us human, and AI is an extension of that quality.*”

Yann LeCun

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

Abstract

Doctorado en Tecnologías de Telecomunicación e Ingeniería Computacional

Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería

Doctor of Philosophy

Contributions to a Methodology for the Building of Modular Neural Networks

by David Castillo Bolado

Modularity is a powerful concept that has been long leveraged by humanity in order to tackle complex problems. The general idea of taking something of significant complexity and breaking it down into simpler parts can be applied to almost anything, but when it comes to software it usually involves the isolation and reuse of functionality. In the field of deep learning, modularity has been widely exploited in the definition of models and learning algorithms, but not so much regarding the isolation and reuse of learned functionality.

In this thesis we delve into the meaning and implications of functional modularity within the field of deep learning, translating in the process some of the most common concepts and design rules to the realm of learned software. With the goal of increasing the degree to which functionality is reused across models, we first analyze the pros, cons and side-effects of training small modules independently, and with that knowledge in hand we then propose a new methodological approach and a series of concepts and tools oriented to design and train highly-reusable modules. Along the document we constantly leverage well-known features of neural networks in order to substantiate key design choices of modular interfaces, which significantly improve their compositional generalization and therefore their reuse potential. We also provide a set of guidelines and methods that contribute to keeping that reuse potential while training the modules in isolation.

Acknowledgements

I would like to first thank my supervisors Mario Hernández Tejera and Cayetano Guerra Artal, for introducing me to the field of deep learning and for their essential guidance and advice. I would also like to thank my lab partners, specially to Pascual Lorente, and Sergio Marrero, with whom I shared countless hours of deep study, fierce debate and coffees. Those days really represent the spirit of learning and research to me. I am also very grateful to all my other lab colleagues Andrea Argudo, Marco Galluzzi, Yeray Gutiérrez, Daniel Hormigo, Pedro Marín, Javier Lorenzo and Modesto Castrillón. They made the laboratory a truly warm place to be, despite what the actual temperature and lighting conditions were. A special mention goes to Antonio Falcón, who sadly left us a while ago, but whose wisdom and sense of humour still lives with us.

I wish to thank my family and friends, who always kindly asked me about my progress and shared my joy with every accomplishment. Huge thanks to my parents, who made all this possible and supported me during my studies. Every new day I become more aware of the priviledge that it has meant to me. I am also incredibly grateful for the love and support that my girlfriend Cristina Juliá has given me during these last two years, throughout all the ups and downs. I really could not have done this without you, Cris.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	**A**rtificial **I**ntelligence
API	**A**pplication **P**rogramming **I**nterface
CNN	**C**onvolutional **N**eural **N**etwork
CV	**C**omputer **V**ision
DL	**D**eep **L**earning
GNN	**G**raph **N**eural **N**etwork
GPU	**G**raphics **P**rocessing **U**nit
MCTS	**M**onte-**C**arlo **T**ree **S**earch
NLP	**N**atural **L**anguage **P**rocessing
NN	**N**eural **N**etwork
VQA	**V**isual **Q**uestion **A**nswering
RL	**R**einforcement **L**earning
RNN	**R**ecurrent **N**eural **N**etwork
SGD	**S**tochastic **G**radient **D**escent
SGM	**S**urrogate **G**radient **M**odule

To my brother Raúl.

As a result of this doctoral thesis, the following scientific contributions have been made.

- A full-paper submitted to the AAAI-MAKE 2019, the AAAI Spring Symposium on *Combining Machine Learning with Knowledge Engineering* (Castillo-Bolado, Guerra-Artal, and Hernández-Tejera, 2019). The symposium took place on March 25–27, 2019 at Stanford University, Palo Alto, California, USA.
- A scientific article submitted to the scientific journal *Neural Networks* (Castillo-Bolado, Guerra-Artal, and Hernández-Tejera, 2021). At the year of publication, the journal had an impact factor of 9.657, ranking Q1 in the categories of *Computer Science, Artificial Intelligence* and *Neuroscience* of the Journal Citation Reports index.

Chapter 1

Introduction

Modularity is an essential feature of engineering. Humans have long leveraged it for building advanced machinery and for understanding and tackling complex problems. Either by recombining existing solutions or by subdividing a difficult problem into multiple, simpler ones, modularity has been present throughout human history. In nature, highly complex systems are also composed of smaller and simpler parts, as it is the case of the human brain. Modularity is, however, a somewhat controversial concept in the field of Deep Learning (DL), as it can have different interpretations depending on the specific application and the assumptions made. Modularity can be applied on many fronts, and so it can be mainly structural (parts being made of smaller pieces) and functional (more complex functions being built by composing simpler functions), and this happens to enter in conflict with the apparent single meaning of the term, which only holds true if we stay at the abstract level and think of "things being made of other simpler things".

Artificial Neural Networks (NNs) are undeniably modular in their structure. Artificial neurons found origin in a simplified model of a biological neuron, just to become the main building block of current DL models. From the very earliest perceptrons to the latest and more powerful models, all of them are built following a modular structure: neurons are organized in layers, which are then stacked together and often used as blocks or modules. Through reduction operations, attention mechanisms and other methods, these modules are glued together in order to build complex architectures, capable of tackling challenging tasks that depend on performing advanced processing of dynamical or multi-modal data. Programming languages and tools that are now ubiquitous allow us to quickly and easily define new models, providing us with ready-to-use instantiators for such commonly-used blocks, with automatic differentiation and GPU acceleration already built in. This turns the exploration of new architectures and end-to-end training into a low-hanging fruit that the field cannot help but keeping delving into, and it probably will not stop doing so for a couple of years at least.

In consequence, the dominating trend right now is to train models end-to-end. In fact, if a model is end-to-end differentiable and has been trained in that fashion, its monolithic nature is most surely presented as an important and beneficial feature. Over and over, state-of-the-art Convolutional Neural Networks (CNNs) rediscover the same visual features just to beat its predecessor[1], and only when training this initial set of layers is an obvious obstacle, a pre-trained CNN is used as a feature extractor. Moreover, there seems to exist now an arms race for who is going to train the next largest model. Initially it only concerned language models (Radford et al., 2018), but the trend quickly spread to the images domain and multi-modal data scenarios (Ramesh et al., 2022; Reed et al., 2022). These now-so-called foundation models (Bommasani et al., 2021) are constantly

[1]https://paperswithcode.com/sota/image-classification-on-imagenet

gathering headlines and attracting the attention of many researchers due to their impressive capabilities and wide range of applications. Authors' claims center around the training scalability of these huge models, praising their ability to keep converging when given more compute time or training data, but what about the scalability in terms of efforts and resources?

Every time a new foundation model is presented, it is also shown that it has been trained end-to-end from scratch. No reused parts or functionality whatsoever. Reasonably enough, this steady increase in resources invested in training large models has become a growing concern (Thompson et al., 2021). In a recent work of Strubell, Ganesh, and McCallum (2019), an analysis of training costs for state-of-the-art NLP models is conducted and it is shown that the amount of energy invested represents CO_2 emissions equivalent to those of a trans-American flight. This analysis also shows that training the model is merely a little fraction of the total development cost, falling most of it into hyperparameter optimization. It is therefore foreseeable that this wasteful training trend might hit a wall at some point in the not-so-far future. Additionally, the history of engineering has demonstrated that having reusable and standardized parts is key for keeping low maintenance costs, increasing the reach and impact of small improvements and facilitating the building of new systems via recombination of existing pieces.

In fact, DL relies already on many standardized layers of modular hardware and software, very often on top of each other but sometimes also intertwined. GPU cards provide low-level APIs that give higher layers access to fast and parallel computation. Automatic differentiation libraries like Tensorflow[2] or Pytorch[3] use these API calls to implement DL primitives, like NN layers or commonly-used

[2]https://www.tensorflow.org/
[3]https://pytorch.org/

loss and activation functions, and also higher-level capabilities like backpropagation and gradient optimizers that allow implementing gradient descent in a few lines of code. Some building blocks are so commonly used that they are also included in these libraries, as it is the case of Long Short-Term Memory (Hochreiter and Schmidhuber, 1997, LSTM) and Gated Recurrent Unit (Cho et al., 2014, GRU) recurrent layers. In terms of code, the vast majority of published models are modular, and making the code accessible to others ensures that it can be reused and that more complex models or methods can be defined. However, it often goes unnoticed that the functionalities and behaviours learned by these DL models are in fact an additional layer of software. This new kind of software component, represented as a large set of real-numbered parameters, is very different from any previous form of software in that it cannot be specified by hand and must be learned instead. As a result, these parameters become obscure to the programmer and DL models are treated as black boxes, making them hard to be altered or repurposed despite the level of complexity of the task it was intended to solve. In this sense, current DL systems are similar to the very first large computers, which could perform tasks unattainable to previous machines but required a team of highly specialized engineers to configure and write code just for a single use-case scenario.

It is therefore the main motivation of this thesis to address the issue of modularity in neural software, properly characterizing essential indicators like coupling and cohesion and giving answer to some key questions that allow establishing an initial methodology. This efforts constitute an initial step towards bringing neural software closer to the standards of written code. In this document, we will take a look at the field of DL through the lens of modularity. We start by reviewing the current state of the art (§2). We place the focus on works related to many different interpretations of modularity, making special emphasis in explicitly modular architectures, modular training and models with compositional behaviour. We then present the experimental works that we have done in this direction. In the first experimental chapter we conduct prospective studies about the low-level technical implications of independent modular training (§3) and we follow up with further analysis and experimentation at the functional and methodological level (§4). Finally, we discuss the main contributions, immediate and future implications of this work, and how it might be continued (§5).

Chapter 2

State of the art

In the mainstream of neural network research and applications, neural networks are viewed as unstructured black boxes. This view is convenient as long as no problems (e.g., with learning speed or convergence) occur. However, we cannot expect this state to persist if our artificial neural networks grow and our applications become more difficult.

Hrycej, 1992

This work's contributions span several areas related to artificial NNs, but also to other areas of AI. Modularity itself is a concept that has been and is still understood in many different ways, depending on the scope and abstraction level. In this section, we provide an overview of the most relevant previous work. We also provide a brief explanation of key concepts that relate to the understanding of modularity within software engineering and the corresponding adaptation of these concepts to NNs.

2.1 Modularity

In general terms, modularity is a structural property that refers to the partitioning of a whole. Something is modular if it can be subdivided in parts, or modules, and this is commonly done in software engineering with the intention of isolating functionality for future reuse, obtaining in the process independent pieces of software. In NNs, as in any kind of software, we can identify several levels at which modularity is present. From the very conceptual level, artificial NNs are modular, since they are composed of parameters or weights, which are grouped in artificial neurons, which are in turn arranged in layers and then stacked up to finally form a deep NN. The automatic differentiation software that is often behind the design and training of NNs [e.g. Tensorflow (Abadi et al., 2016) and Pytorch (Paszke et al., 2019)] is also modular: core functions and elements of artificial NNs share code, keeping the whole project easy to develop and maintain. In this work, however, we focus on the modularity that exists on top of all these layers, in the form of neural modules or modules whose functionality is implemented by a NN.

As a structural property, modularity does not provide any guarantee of functional independence (Béna and Goodman, 2021). What the actual implementation of the module looks like, how it establishes communication with other modules and what functionality it implements are crucial factors that will ultimately determine the generalization and reuse potential of the module. In this regard, coupling and cohesion are two important concepts of software engineering that characterize how good a module's functionality and interface have been defined:

1. Coupling is a measure of the degree of interdependence between modules and it is therefore recommended to keep it as low as possible. Loosely coupled modules perform always the same function, regardless of what modules it is interacting with and do not alter any external data or produce side-effects. Highly coupled modules are often functionally brittle and hard to reuse and maintain.

2. Cohesion characterizes how much the information elements contained in a module are functionally related. It measures the level to which it makes sense that some information is stored or treated in the module and not somewhere else. It is therefore beneficial that modules have high cohesion.

If we translate these concepts to the field of NNs, coupling is now a multi-modular expression of overfitting, in which several modules overfit to having been trained jointly, and lowering the inter-modular coupling means minimizing the probabilities of this overfitting taking place. Cohesion is also related to overfitting, since it decreases when the module stores information that it should not store, and this is a especially difficult thing to keep under control when modules are trained jointly.

Motivations for modularity in the field of NNs have stayed more or less the same across the past years. Under the motto of *'Divide and Conquer'*, the desire of subdividing a complex monolithic model into simpler components has been the common driving force. Hrycej (1992) enumerates four main reasons of different nature for seeking modularity:

1. *Engineering.* From an engineering standpoint, having a modular system with intermediate solution stages that can be analyzed and validated is

strongly preferable to having black-box monolithic systems.

2. *Complexity.* The more complex a NN is, the harder it is to train. Standard training tools like backpropagation quickly fall short when model complexity grows.

3. *Psychological aspects.* In particular the observation of different learning stages and the incremental nature of human learning.

4. *Neurobiology.* The human brain has distinct functional areas, which can be anatomically different and exhibit functional specialization. These parts communicate but are functionally independent.

The design of modular NNs has always been conducted following three main stages: task decomposition, training of modules and multi-module decision making (Auda and Kamel, 1999, Figure 6). However, the interpretation of these stages differs significantly from what we describe in this document. During the 90's and early 2000's there was a commonly accepted equivalence between task-decomposition and the delegation of parts of the input space to different modules, which led to the exploration of many kinds of (automatic or systematic) model ensembling while overlooking other forms of module combinations that could implement functional composition. A notable exception to this trend is the work of Waibel (1989) on merge-and-glue networks applied to speech recognition, where he pre-trained distinct modules on different sub-tasks and then *glued* them together, forming a more complex NN capable of outperforming monolithic approaches in the global task. Merge-and-glue networks are one of the first examples of modular-wise supervised training and they showed how it was possible to decrease the computational cost of training a NN through this principle. During that time, however, modular recombination and knowledge reuse were not among the main concerns and therefore they were not addressed.

The widespread view of modularity from a task-subdivision perspective focused efforts on finding generic sub-tasks, which led to finding different kinds of decompositions of the training procedure. For example, by identifying different learning types (supervised, unsupervised, etc.), input mappings (linear and non-linear) or by separating learning into a knowledge-based part and a learning part (Hrycej, 1992). Most of the research was focused on speeding up the computation and easing the implementation and training of neural layers in deep feed-forward networks. In contrast, more recent taxonomies classify modular networks into tightly and loosely coupled networks (Chen, 2015), which is more in line with our view, albeit only displaying ensemble-like networks under the loosely coupled group and not NNs capable of functional composition and compositional generalization. Additionally, in that same survey, some past interpretations of modularity like the brute-force conversion of multi-class problems into many bi-class problems (Lu and Ito, 1999) are reinforced.

Nevertheless, many observations were made that are very important with respect to our contributions. In relation to the training procedure, inter-modular crosstalk (Jacobs, Jordan, and Barto, 1991) was identified as an important challenge and inter-modular coupling was analyzed, although at a very low level (between neurons) (Auda and Kamel, 1999, Section 2.4.1). This view of coupling contrasts with our analysis of inter-modular coupling at the task level (see §4). Another interesting observation made in previous research is related to Mixture of Experts (Jacobs et al., 1991, MoE), in which it was found that functions learned by the modules should be fundamentally different, since otherwise all functions would be learned collectively by all modules and the possible benefits of modularity (even from the input-space partition point of view) would be wasted (Wang, Nasrabadi, and Der, 1997). Finally, hardware motivations led to the development of currently ubiquitous tensor computing and automatic differentiation libraries, like Tensorflow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019).

Industrial applications, on the other hand, have always tended to implement NNs following a high-level task decomposition, mostly within the area of automated control systems. Some early examples can be found in air conditioning (Kah et al., 1995) or autonomous driving systems (Mecklenburg et al., 1992). Hrycej, 1992 identified the modularization of application tasks as *"one of the most powerful modularization approaches"*, and these examples are certainly a first step in that direction, but all their modules are designed ad-hoc for the task at hand and can hardly be reused in future models, let alone in different tasks.

Also motivated by the optimization of research efforts, many new building blocks exist now that allow to quickly design and train monolithic NNs. These building blocks have also suffered a good amount of standardization, given that they are provided as part of the mainstream frameworks[1]. Convolutional layers (LeCun et al., 1989), residual blocks (He et al., 2016), self-attention layers (Vaswani et al., 2017) and more complex blocks like LSTMs (Hochreiter and Schmidhuber, 1997) are among the most used architectural components. This, in conjunction with the ease of use and flexibility of current frameworks for NNs design and training, has led to an explosion in the exploration of new architectures and training methods (Szegedy et al., 2015; Zoph et al., 2018). Having these building blocks easily available is one way of improving the design capabilities

[1] https://pytorch.org/vision/stable/models.html

of the research community, by putting such a wide range of carefully designed solutions at everyone's disposition. This is, in fact, a democratization process of structural and architectural modularity, and yet it only solves part of the problem, since the functionality of these parts is left to be acquired from scratch through end-to-end training.

Many modular design patterns in NNs are actually motivated by the desire of increasing the amount of weight sharing, which is known to improve training efficiency and generalization. Weight sharing represents a way to integrate expert knowledge, as it explicitly guides training towards reusing knowledge in very specific places, implying that the shared weights are useful for those cases in which they are applied (LeCun et al., 1989). This aspect of weight sharing has made it a common element in NN architectures with a high degree of modularity and inductive bias, like CNNs (He et al., 2016), Recurrent Neural Networks (RNNs) or Graph Neural Networks (GNNs) (Shlomi, Battaglia, and Vlimant, 2020), among others. This kind of inductive bias is often related to the concept of invariance, being spatial invariance (e.g. in CNNs) and order invariance (e.g. attention mechanism in graph convolutions)(Zhang et al., 2019) the most common types, and it is a key concept towards defining functional modularity. It is also one of the key factors that drive the emergence of the *Learning by Role* phenomenon (§4.2.2).

Notable results have been achieved recently by exploiting the modularity of existing algorithms (Dai, 2020). In particular, this line of research leverages the fact that every part of an algorithm that relies on heuristics or abstract rules is subject to being successfully implemented by a NN. Among the most notable examples there is the steady-state operator in the Stochastic Steady-state Embedding algorithm for low-cost embedding of large graphs (Dai et al., 2018). Advances in this line with most impact are characterized by learning the exploration heuristic of search algorithms in zero-sum games (Silver et al., 2016; Silver et al., 2018) and augmenting the search with learned reward-oriented world models (Schrittwieser et al., 2020). These results are examples of the impact of local improvements in decoupled modular systems, in which parts benefit from that local change and the whole system's performance increases, instead of suffering due to local overfitting issues. Because common algorithms are already modular and have discrete and well-designed interfaces, changing the implementation of one part of it does not hurt the whole as long as the altered part keeps the original goal or main functionality. Even if interfaces do not help, because these algorithms are made of parts that we already understand, we can easily predict the consequences of local changes and prevent unwanted side-effects.

A key feature of generalizing modular architectures is the modularization of the global task into primitive operations and many levels of abstraction. We can already spot this phenomenon in deep NNs, in which neurons specialize in detecting many diverse features and every layer builds on top of previous layer's features, thus detecting higher-level features and giving rise to a hierarchy of abstractions. Surprisingly, this hierarchical organization emerges naturally from end-to-end optimization of deep NNs and it largely explains the capacity of DL models (Yosinski et al., 2015). However, and as good as this characteristic is, it does not allow for easy reuse nor compositional generalization due to many local overfitting factors. First, the scale of the activations greatly changes across models, even when sharing the same architecture and just having been trained on different samples. Second, the ordering of those activations also changes depending on many stochastic factors, like weight initialization, sample ordering or the choice of a different optimization algorithm. Third and finally, current implementations of neural layers have a fixed number of inputs, which might be reduced after training through pruning (setting weights to zeros), but not increased. In regard to modularity in static architectures like those of common deep NNs, a concurrent work has revealed that some clusters of neurons in DL models exhibit aspects of modularity (Hod et al., 2021). They identify coherence, among other measurable features, and they also distinguish structure from functionality by defining modularity in terms of how parts of the computational graph (structure) can be said to perform some comprehensible sub-task which is relevant to the overall task (functionality). This definition is pretty much in line with our current understanding of modularity and differs from the previously mentioned early definitions of it.

An interesting way to avoid input order and size limitations are attention mechanisms (Vaswani et al., 2017), although they require for any kind of ordering information to be embedded in the input elements and they do not solve the overfitting to the input scale. Neural Module Networks (Andreas et al., 2016, NMN) and all its variants exploit both attention mechanisms and the use of primitives in the search of compositional generalization, mainly in the realm of Visual Question Answering (Antol et al., 2015, VQA). NMN rely on a high-level partitioning of the task into primitive sub-tasks, which are implemented by modules that extract very well defined features and that are trained for maximizing their reusability. More specifically, they implement the task of answering a visual-related question by combining three primitives: locating an object in the image, answering a question in relation to the objects' size or shape and answering a question about the object's non-spatial features. Additionally, they leverage hard-attention as well as soft-attention mechanisms for setting up the different modular layouts in consonance with the input question and combining the results produced by the different modules. Significantly, they also leverage the pre-existing Stanford parser (Klein and Manning, 2003) in order to inform the construction of modular layouts and instantiate the object-finding modules. Differently from the early notions of multi-module decision making, which mainly set a homogeneous input and output space, all of these neural modules implement different functionalities and they make explicit use of distinct representational spaces, which resembles the neural data types proposed in this thesis (see §4.2.1). They are not targeted for diversity or mere feature extraction purposes, but for accomplishing very primitive tasks whose results can be fed to other modules and composed in complex ways in order to complete a higher goal. However, and as we will see in detail in chapter 4, NMN only show in-distribution compositional generalization and present multiple sources of task overfitting and inter-modular coupling that prevent it to generalize well to novel layouts.

2.2 Modular training

The training of NNs has been typically done in a task-oriented manner, being the most common goal to minimize some form of expected error between the currently modelled function and the targeted one. Notably, any thinkable task is potentially a subtask of another, more complex task. This situation has of course led to many NNs being reused in tasks other than what they were originally intended to solve, ending up becoming a functional module within a larger NN. The reuse of neural models can be and often is purely architectural, in which cases the trained weights are simply discarded and the network is trained again, from scratch but in close interaction with the new architecture and its training goal (Alom et al., 2019). It can also be in the form of an initial set of weights, which is then fine-tuned for the new task (Hong, 2022). Another possibility, more in line with the concept of modular training, is that the repurposed NN is reused *as-is*, taking also the pretrained weights and integrating the network into the new architecture's pipeline (Mao et al., 2019). In this latter case, however, the pretrained network typically plays a role at the very beginning of the pipeline, becoming therefore just another stage of the input pre-processing procedure (Hu et al., 2017). Another common case is when the network is a world model and its value resides not only in its functionality or knowledge but also in its differentiablility (Lutter et al., 2021), so that it is worth placing it somewhere else along the pipeline. This particular use case leverages the world model during backpropagation, letting it shape gradients and thus guide the training of the network through them. Implicitly, the world model becomes part of the loss function for a portion of the entire network, a use that holds some similarities with the Surrogate Gradient Module (SGM) proposed in section 4.2.3. In general, when we look into the literature, the urge to have end-to-end training capabilities tends to prevail over re-purposing NNs.

In any case, when NNs are simply re-purposed, issues with the scaling and processing of input and output values often arise. This is merely a consequence of these networks not being intended for reuse and just being designed and trained with the goal of achieving a new state of the art in some specific task or benchmark. In contrast, we aim to have neural models that are designed and trained with reuse in mind, explicitly orienting them to learn highly useful subtasks and therefore becoming potential parts of many other models.

In what relates to training parts of NNs or having NNs be trained for later use, most previous work can be found under unsupervised learning. Unsupervised learning methods, like Hebbian learning (Hebb, 1949) or self-organizing maps (Ritter and Kohonen, 1989), are at the core of the earliest forms of modular training. They allow pre-training of neural modules without the need for labels, learning this way good non-linear representations of the data or at least finding good initial weight configurations. On top of such good non-linear representations, well-known and robust linear methods can be often applied in order to obtain good classification or regression results, which is in practice similar to the kernel trick (Theodoridis and Koutroumbas, 2006) but relatively inexpensive to compute. Recently, self-supervision methods have seen significant success in Natural Language Processing (NLP), allowing to pretrain highly accurate language models from huge amounts of unlabeled text (Devlin et al., 2018). This kind of methods are also present in CV, mostly found under the term *contrastive learning* (Gidaris, Singh, and Komodakis, 2018; Chen et al., 2020), and have been also applied to tabular data (Yoon et al., 2020). Self-supervision differs from classical unsupervised learning in that it often requires some degree of knowledge about the data manifold (e.g. augmentations to images in the form of rotations or random crops), which enables using part of the data as target labels. In addition, self-supervision methods rely on end-to-end backpropagation and are therefore global, in contrast to more knowledge-agnostic methods like Hebbian learning, which are purely local. In any case, models that have been trained with unsupervised or self-supervised methods are highly task agnostic and they can be used for improving performance on tasks for which very little data is available (Artetxe and Schwenk, 2019). However, being only able to work in the scope of representation learning has its limitations and these models remain still a black box which is nearly impossible to divide into parts that can be later reused.

In contrast, looking into industrial applications, many solutions have long relied on fully-supervised training of neural modules, requiring significant design efforts but also being highly reliable (Kah et al., 1995). These industrial solutions are very problem specific though and the trained parts of the models are rarely repurposed, since they are trained with very particular and expensive data. Back into the academic field, Reed and De Freitas (2015) showed that using fully-supervised traces allows the network to interact with non-differentiable elements. They leverage this ability by means of scratchpads, which form part of the environment and can be used for storing information, but more importantly for keeping a stack of hidden states and thus allowing calls to subprograms. This particular mechanism has been proved extremely useful for exhibiting compositional generalization and it has been used by Cai, Shin, and Song (2017) for implementing recursion and providing generalization guarantees for the first time in the field of neural program induction. Some authors have also tried to jointly train models with highly modular architectures (Gupta et al., 2019), which have nearly discrete interfaces, but they have found that the discrete nature of these interfaces is often in contraposition to the differentiability that end-to-end training requires. In consequence, they rely on modular auxiliary losses too, which are based on heuristics about known or desired features of the corresponding intermediate values.

In many successful modular NNs, end-to-end training is used for training the full model, and then modules acquire their behaviour as a side-effect of this process (Andreas et al., 2016; Shlomi, Battaglia, and Vlimant, 2020). This phenomenon, that we call *Learning by Role* (see §4.2.2), is thus responsible for embedding skills or functions into parameterized modules and it can be therefore seen as a method well suited for modular training. However, and to the best of our knowledge, it has never been used as a tool for training a single module in isolation. This is, without the corresponding module being part of an architecture of higher-complexity that is trained in an end-to-end fashion. *Learning by Role* is also related to meta-learning and learning-to-learn, where a similar technique is used for optimizing the parameters of the learning rule, which is expected to converge towards an optimization policy. Thanks to the information provided by gradients that come from a higher-level meta-loss, the learned optimizer is able to adjust its parameters and approximate a function that has never been specified in a direct manner, but through the role that the learned optimizer plays in the inner loop (Andrychowicz et al., 2016). This way of leveraging gradients for the indirect and implicit shaping of a module's behaviour is nevertheless significantly complex and entails several issues related to the level of chaos and optimization difficulty of the outer-loss landscape that results from unrolling the inner loop (Sohl-Dickstein, 2021).

2.3 Modular architectures

Although structural modularity does not provide any kind of guarantee that the network will exhibit compositional behaviour (Béna and Goodman, 2021), there are some modular architectures that are worth mentioning due to the motivations behind their design and their use of inductive biases, in many cases actually leading to highly compositional behaviour (Battaglia et al., 2018). In this section, we will review some neural architectures that can be said to be modular, for they have been conceived with a certain structure in its functioning, regardless of the final organization of the knowledge acquired during training.

One of the most primitive forms of architectural modularity in NNs is weight sharing (Rumelhart, Hinton, and Williams, 1985), which is an inductive bias that leverages structural regularities in the data for increasing parameter efficiency, therefore reducing the number of free parameters in the network, boosting training and improving generalization. Weight sharing represents a structural form of knowledge embedding, by which the designer knows in advance which parts of the network should share functionality and forces such setting by using the same parameters in the corresponding places. This way, the training is transformed into a constrained search and the parts being reused benefit from richer learning signals. Examples of weight sharing are to be found all over the field: RNNs leverage the sequential dependence of inputs for sharing weights across different steps, CNNs do so across different spatial positions and GNNs share weights across nodes and edges. Sharing weights turns the reused parts into many instances of the same functional modules, and this improves generalization and also accelerates training by aggregating sparse learning signals and concentrating it into few key points. As a direct consequence of the training, weight sharing results in compositional behaviour, but this behaviour can only be guaranteed as long as samples are from the training distribution, often resulting in performance degradation when the presented combinations are slightly different [e.g. longer sequences for RNNs (Liška, Kruszewski, and Baroni, 2018) or larger graphs in GNNs (Sanchez-Gonzalez et al., 2018)].

Relation Networks (Santoro et al., 2017) are neural modules designed with weight sharing at their core for explicitly representing the combinatorial bias in relational tasks. Object representations are concatenated in all possible combinations and a query embedding is included for informing the feature gathering process, which is done by the same MLP in every case. The resulting feature vectors are summed together and passed through a final MLP that gives out the answer. Although computationally expensive during inference, the module can generalize well from little training data. This approach can be understood as the inference-time equivalent of data augmentation, which explores the combinatorial space during training time (Shorten and Khoshgoftaar, 2019; Andreas, 2019). However, in the original paper the authors do not exercise the composition of these modules and they instead combine their outputs through aggregation. On the other hand, both the information-gathering process and the quadratic cost of this module establish a direct connection with Transformers.

Transformers (Vaswani et al., 2017) represent an architectural pattern by which all elements of a sequence of arbitrary length are processed in parallel and whose elements are subject to a self-attention mechanism. By means of this processing pipeline, the representation of the input sequence is transformed after the application of each layer, but the sequential organization of the information also changes layer after layer. Thanks to each element in the sequence having its own key-value pair, a potentially singular search is conducted for each input token, but most significantly that search is driven by a latent function that is shared across all elements in the sequence. Transformers have revolutionized the world of NLP by heavily relying on the modularity and combinatorial power of self-attention, and they have showed how such mechanisms enable these architectures to grasp the complexity of human language (Brown et al., 2020). The interpretation of Transformers from the point of view of the individual elements in the sequence has driven some authors to compare them with a population of collaborative agents that share the same policy (Rosa et al., 2019), which has also led to the proposal of newer and more versatile architectures that dramatically increase the level of weight sharing through recurrence (Ontañón et al., 2021) or to get rid of the dimensional constraints of self-attention in favor of cross-attention and multimodal data (Jaegle et al., 2021). In the case of the Perceiver (Jaegle et al., 2021) it is particularly straightforward to see how memory positions can be interpreted as different memory states and how each position is updated by gathering information from different positions in the input. This update is therefore unique for each position, despite having been done by the same policy.

Recent architectures that have gained public attention due to their performance and surprising abilities count with external memory modules (Graves et al., 2016; Thoppilan et al., 2022). It has been shown that relying on non-parametric modules for storing temporary or supplementary information not only improves the network's performance by relieving it from learning the remembering mechanisms, but also boosts generalization, specially if the memory interfaces promote interpolation and local interactions (Joulin and Mikolov, 2015; Grefenstette et al., 2015; Kurach, Andrychowicz, and Sutskever, 2015). However memory modules are often highly complex in order to keep full differentiability and allow end-to-end training, which also tends to result in coupling issues (inter-modular overfitting). These issues have been tackled so far by seeking sparse interactions (Goyal et al., 2019), which also have positive impact in compositional generalization and have been proven to do so (Béna and Goodman, 2021).

Among the architectures that explicitly define modules with strongly differentiated functionalities and diverse inference-time compositional layouts, Neural Module Networks (Andreas et al., 2016, NMN) stand out. They specify a set of neural modules, each one of them intended to have a distinct functionality and therefore provided with a proper ad-hoc architecture, which is different in each case. Depending on this envisioned functionality, the module's input-output interface might vary and this makes them not always be compatible, which is in contrast to Kirsch, Kunze, and Barber (2018), where they rely on homogeneous modular architectures and therefore interfaces that are always compatible. This kind of compatibility constraints resemble data types and are indeed responsible for heavily reducing the search space of modular layouts. However, they did not explore other type-related aspects of modular interfaces (see §4.2.1) and their modular layouts were always extremely shallow. Later implementations of NMN focused on explainability and gave up on investigating the discrete composition of modules in favour of differentiability, which introduces inter-modular coupling and therefore hurts generalization to larger layouts (Hu et al., 2018). NMNs have also been extended to cover text-related tasks (Gupta et al., 2019), which required the introduction of new and more diverse functional modules and the use of modules with text-extraction and indexing capabilities. These functionalities challenge the differentiability of modules and forced the authors to rely on auxiliary losses, which were set at intermediate places in the layouts for facilitating end-to-end training.

Still in the area of VQA, Mao et al. (2019) propose the Neuro-Symbolic

Concept-Learner (NS-CL), which also exploits the discrete composition of modules. As an interesting improvement, they leverage a CNN pre-trained on a segmentation task for identifying objects in the scene and extracting high-quality object-level representations. This leap from raw images or features to whole objects sets up a highly compositional search space of possible programs. This quasi-symbolic execution, as they call it, retains a resemblance with the MCTS used in models like AlphaGo (Silver et al., 2016), which also benefits from a discrete search space and an end-to-end differentiability at the same time. However, these representations are still vectors with unbounded real-valued numbers, so they face very similar coupling issues than those of GNNs. As Andreas et al. (2016) and Alet, Lozano-Pérez, and Kaelbling (2018) do, the authors rely on a combination of backpropagation for learning the modules and a version of combinatorial optimization for learning the composition of modules. They additionally highlight that using curriculum learning is essential to help find good early configurations for the modules, so that the combinatorial search works better later on and does not cause learning interference (Hu et al., 2017).

2.4 Compositionality

Although compositionality can usually refer to the action of composing or combining parts, in the field of NNs it is more frequently used for describing the ability of a system to exhibit compositional behaviour. This is to say that a system can show some behaviour which is the product of combining several simpler behaviours, with the goal of producing combinatorial generalization or the *'infinite use of finite means'* (von Humboldt, 1836/1999). Other equivalent definitions given in the literature for compositionality are *'the algebraic capacity to* *understand and produce novel combinations from known components'* (Montague, 1970) and *'the ability to recombine meaningful units with regular and predictable outcomes'* (Loula, Baroni, and Lake, 2018).

It is therefore worth highlighting the additional value of re-composition, since we can already find parts or skills being reused in every NN [reusing features during the abstraction process of the different layers (Yosinski et al., 2015)], and it is precisely the ability to recombine modules in novel ways what leads to compositional generalization. Multi-task learning (Caruana, 1997) is one field in which skill reuse has played a key role as the main favourable feature, which enables leveraging the commonalities between the tasks (positive transfer) while minimizing interference (negative transfer).

On its own, compositionality does not impose any restriction regarding modularity. It is so that in the field of NNs there are several cases in which a monolithic network exhibits compositional behaviour to some extent, coexisting this behaviour with some degree of ad-hoc memorized responses (Yosinski et al., 2015; Andreas, 2019; Hupkes et al., 2019). Modular systems, on the contrary, advocate for explicit compositionality and implement it via the modular inductive bias. In modular NNs, the connections between different isolated parts or modules cannot be formed freely (like in neural layers), but can only happen through the modular interfaces and according to the modular layouts imposed by some form of control entity. This modular inductive bias aims to provide the whole system with low coupling and high cohesion, since the sparsity of connections is strongly related to functional specialization. According to Béna and Goodman (2021), structural modularity does not guarantee any degree of functional specialization, but extremely sparse and low-bandwidth interfaces do induce functional modularity. The reasoning behind this relationship is that inter-modular coupling tends to rely on dense interconnections.

Hupkes et al. (2019) provide a series of tests oriented to evaluate the degree of compositionality of NNs. These tests are of special relevance because they do not care about the internals of the model, but instead they observe and measure the model's behaviour from a black-box perspective. This way they provide both an accurate definition of what compositionality is and a universal means for testing it. The tests they propose focus on five key properties of compositional generalization:

- *Sistematicity* evaluates the model's ability to recombine known parts to form new sequences e.g. *"Someone who understands 'brown dog' and 'black cat' also understands 'brown cat'."* (Szabó, 2012).
- *Productivity* focuses on the model's ability to understand sequences longer than those seen during training.
- *Substitutivity* evaluates how robust the model is to replacements of elements with other equivalent or similar elements.
- *Localism* measures how local the model's compositional operators are i.e. whether local rules precede more global ones.
- *Overgeneralization* tests how likely is a model to accommodate an exception instead of following the general rule.

This seminal work is essential for understanding the underlying motivations of compositional interfaces and architectures.

The very same model-agnostic notion of compositionality can be applied in the opposite direction too, that is, in order to train monolithic NNs for showing some degree of compositional behaviour. Instead of hard-coding compositional inductive biases in the network's architecture, one very effective alternative is to make the network explore a diverse variety of input combinations during training. This method, called data augmentation, exploits known symmetries present in the input data to generate alternative views of the training samples (Tanner and Wong, 1987). Data augmentation methods, very widespread in the field of image recognition (Shorten and Khoshgoftaar, 2019), have been recently applied in NLP in an attempt to brute-force the learning of the compositional relations in the input (Andreas, 2019; Akyürek, Akyürek, and Andreas, 2020). Although significant success has been reported, the computational cost is shifted into training time and models still suffer from severe loss of generalization at arbitrary input lengths or complexities. Therefore most NNs under data-augmented training are probably just benefiting from a richer training stimulus and can only exhibit as much combinatorial extrapolation as their architectures allow for.

Coinciding with the observation that language is highly compositional and that it exhibits combinatorial generalization, significant compositional generalization has been observed in some language models with self-attention mechanisms (Radford et al., 2018; Ontañón et al., 2021). The key resides in the ability of attention mechanisms to accept a variable number of input elements while being order invariant. Recently, this ability for compositional generalization has been exploited in the realm of image generation (Ramesh et al., 2021) and as a way of building a structure-agnostic perception module through iterative application of composable modules (Jaegle et al., 2021). In this kind of attentional models, however, performance still drops when the number of elements grows or the complexity of interrelations increases, pointing to a fundamental flaw of the internal compositional mechanisms, which somehow overfit to those properties. In section 4.2.1 we give some hints that help explaining this aspect.

Intuitively, compositional behaviour seems to be reproducible through the recombination of a finite set of composable modules. However, and as pointed out before, most contributions in the literature only focus on models exhibiting this ability during learning or in relatively controlled conditions [e.g. limiting the modules to be into one very particular set (Kirsch, Kunze, and Barber, 2018)]. Routing networks, for example, try to learn modules and a policy that combines them at the same time (Rosenbaum et al., 2019), and it ensures the inter-modular compatibility by making all modules' inputs and outputs be of the same size. This approach refuses independent training and argues that everything must be learned jointly. However, the routing of modules is done via hard choices and learned through reinforcement learning, which means that this architecture holds –at least in principle– the possibility of using more appropriate and composable modular interfaces. In fact, Abolafia et al. (2020) train through reinforcement learning a neural composer for combining a set of pre-defined modules, with the goal of inducing a range of algorithmic tasks from input-output examples (algorithm induction). Not caring about module-through differentiation, they leverage discrete, fixed and relative representations. As a result, they have a memory-less controller which has very good generalization properties, exhibiting perfect generalization in some tasks.

In a similar way than how modules' outputs are fed to other modules' inputs in routing networks, recurrent NNs learn to receive part of its own output as input. Especially if we unroll the feedback loop of the network we can see that this setup is equivalent to a sequence of concatenated modules which share their weights[2]. Through weight sharing and with the right inductive biases, these kind of networks are able to exhibit some degree of (sequential) compositional generalization (Hochreiter and Schmidhuber, 1997). Although theoretically possible, it has been shown that RNNs do not tend to learn compositional generalization (Loula, Baroni, and Lake, 2018; Liška, Kruszewski, and Baroni, 2018). On the contrary, Transformer networks do seem to be better suited for learning to recombine information, but they usually have a fixed number of layers, each of them with different parameters. They can nevertheless work as recurrent networks just by incorporating weight sharing across layers (Jaegle et al., 2021; Ontañón et al., 2021).

Perfect generalization results were achieved by Cai, Shin, and Song (2017) when implementing recursion into Neural Programmer-Interpreters (Reed and De Freitas, 2015). Recursion significantly improves generalization by bringing the problem complexity down to a couple of cases (reductions and base cases). But the generalization is also allowed by the interaction with discrete elements, which eliminates any possibility of error accumulation, and by the restriction of the reception field to a fixed size, as seen in Abolafia et al. (2020). Storing the programs as embeddings is therefore the remaining issue, as it not only uses continuous non-bounded representations, but also requires joint training in order to avoid forgetting previously-learned programs. In this sense, a very important prior work is that of Li et al. (2020), which shares with this thesis many fundamental concepts about the importance of input and output interfaces for attaining

[2]https://colah.github.io/posts/2015-08-Understanding-LSTMs/

compositional generalization. They show that a careful design of interfaces, with fixed-size and discrete values, enables perfect generalization and, in combination with imitation and reinforcement learning, allows for finding very efficient solutions and even improving over existing algorithms. Especially, the combination of reinforcement learning and discrete interfaces seems to enable extremely complex recursion paths in the architecture without having to cope with accumulating errors. In summary, they seem to have a good grasp of the underlying phenomena that motivate our proposed neural data types (§4.2.1).

Chapter 3

Modularization from scratch and side-effects of modular-wise training

When it comes to building a NN with a target task in mind and oriented to end-to-end training, there are already established architectural patterns and guidelines, which greatly reduce the amount of architectural decisions to be made and increase the potential performance of the network. There are however no such established frameworks for modular NNs, which makes sense given the absence of modular NNs in the field that are not trained end-to-end[1]. The reuse of previous knowledge in the field of DL is commonly limited to architectural choices and training methods, and only sometimes practitioners rely on transfer learning and fine-tuning as a somewhat inconvenient way to build on top of previously distilled knowledge.

In this chapter, we address the design of a modular NN from zero, implementing neural modules that can be reused, replaced and improved over time.

[1]With the notable exception of the *Merge-and-glue networks*(Waibel, 1989).

We take this approach to modularity not only for methodological and standardization reasons, but also as a way to study the side-effects of training modules independently and how the modular-wise training affects the final performance and compositional generalization.

3.1 A reuse-oriented architectural framework for modular networks

Our proposal is an architectural framework inspired by the black-board design pattern (D. Erman et al., 1980) and based on a perception-action loop (Figure 3.1), in which the system is an agent that interacts with an environment via an interface. The environment reflects the current state of the problem, including auxiliary elements such as scratchpads, markers or pointers, and the interface provides a representation $R(t)$ of it to work with. $R(t)$ is meant to be a sufficient representation of the environment's state at time t. This representation will immediately reflect any relevant change in the environment and vice-versa, being able to forward changes back to the environment if such changes were made by the agent in the representation. This feedback through the environment is what closes the perception-action loop and it is something that has been seen to work well for memory-less controllers in the past (Li et al., 2020).

A control module decides, conditioned on the environment's representation and its own internal state, which action to take at each time step. These actions are ultimately made effective by operators, which have a uniform interface: they admit an environment representation as input and they output a representation as well. They can therefore alter the environment and they will be

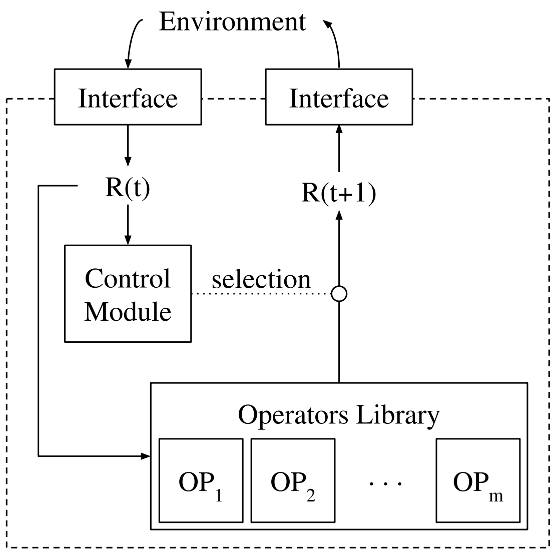


Figure 3.1: Perception-action loop. Each module is susceptible to being implemented by a NN. At each step, the control module selects an operator to be applied and this will generate the next environment's state.

used by the control module to do so until the environment reaches a target state, which represents the problem's solution. As seen in Figure 3.2, each operator is composed of a selective input submodule, a functional submodule and a selective update submodule. Both selective submodules act as an abstraction layer in order to help decouple the functionality of the operation from its interface, minimizing as well the number of parameters that a neural functional submodule would need to consume the corresponding input.

Please note that we do not impose any prior restriction regarding module implementations and therefore the architecture allows the building of hybrid systems. This has significant implications concerning maintenance and knowledge embedding, in the form of reutilization of existing software, manual coding or supervised training of modules. A system that makes use of these modules might improve over time via replacement or addition of modules. Also note that, if

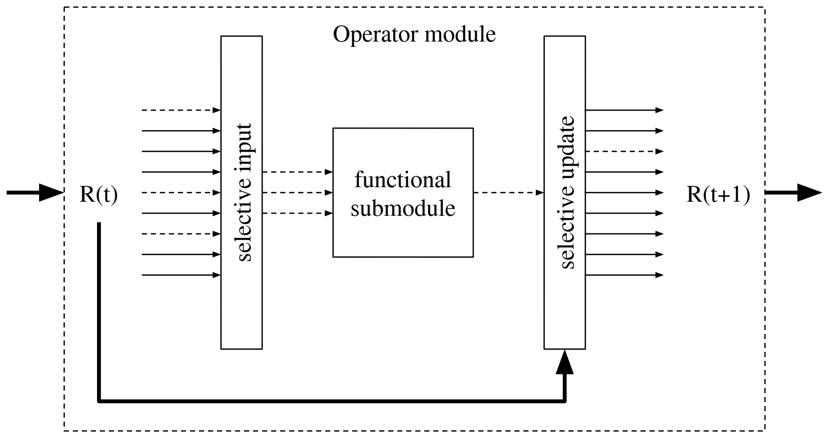


Figure 3.2: Detail of an operator module, composed of an input-selection submodule, a functional kernel and a selective-update submodule. Dashed lines highlight the selected data.

modules are added, the control module must be updated in consequence.

3.1.1 Motivations and elements

This simplified modular architecture is mainly motivated by the desire of having a test framework in which to evaluate the effects of independent modular training in an architecture that is already modular. All problems can be decomposed in subproblems and thus a solution to a problem's instance may be divided into a sequence of primitive operations, each of them transforming slightly the problem's state and bringing it closer to the solution. The scenario –widely accepted in reinforcement learning– in which an agent takes an action based on its perceptions in order to reach a certain goal, inspired us to think about problem solving in these terms. The intention of increasing maintainability of modules by means of a low coupling is also reflected in the modular design.

In the following, we introduce the main components of the architecture and we elaborate on their role in the system:

- *The environment* represents the state of the problem and contains all information involved in the decision making process. The environment is rarely fully available to the agent, so it is generally just partially observable.
  - *The environment representation* is an abstract representation of the environment, which aims to be sufficient for estimating the true state of the problem and therefore for taking the optimal action.
  - *The interface.* Its role is to keep the environment and its representation synchronized. Any changes occurring in the environment are reflected in its representation and vice-versa, therefore keeping the agent abstracted from the environment while allowing fluent interaction.
- *The control module* is the decision making module. It selects which operation should be executed next, according to the current observation of the environment. This module may be seen as an embodiment of the agent, being the operation modules the tools that it uses for achieving its goal.
  - *The digestor or perception module* takes an environment representation as input, which may have unbounded size and data type, and generates a fixed-size embedding from it. It therefore acts as a feature extractor for the policy.
  - *The policy* decides which operation to execute, conditioned on the fixed size embedding that the digestor generates.
- *Operation modules* implement the primitive operations that the agent has available for tackling the problem. Their architecture focus on isolating

functionality, while at the same time allowing interfacing with the environment representation.

- *The selective input submodule* filters the environment representation to select only the information relevant to the operation.
- *The functional submodule* implements the primitive operation's functionality.
- *The selective update submodule* uses the output of the functional submodule to update the environment representation correspondingly.

In general, the intention behind this architectural framework is to give an example of how modular design can strongly delimit the coupling between components, therefore maximizing the probability that a component can be simply swapped out and improved or replaced without further complications. Often, only adjacent components would require slight adjustments for accommodating the upgrade.

While the control module is clearly the one with the highest degree of coupling with the task, perception modules can be reused among tasks in the same domain and functional modules are basically task agnostic, so they can be reused indefinitely.

Additionally, we would like to point out a feature that we think is very interesting and has great potential, which is the possibility of implementing a hierarchy and even recursion within this framework. In other words, an operation module might well be an encapsulation of another agent, whose task is to carry out some functionality that solves a sub-problem for the main agent. This sets an equivalency between the environment's interface and the selective submodules, and opens the door to establishing arbitrary hierarchies between different agents, also including self calls for recursive problem solving. We see this as a fundamental advantage of the modular approach.

3.2 Experiments

With the goal of putting this framework to the test and investigating the effects of modular-wise training on the final performance and compositional generalization, we conduct a series of experiments on a specific implementation. For that purpose, we take a case study based on a list-sorting problem. The code implementing the experiments presented in this section is publicly available at `https://gitlab.com/dcasbol/nn-modularization`.

3.2.1 List sorting as a case study

We take the problem of sorting lists of integers as a minimal case for analyzing the various effects that independent modular training might have. In this regard, we were mainly interested in having a problem that was complex enough to require the use of multiple primitive operations, but not too many of them or too complicated. Additionally, we wanted to be able to measure instance complexity and list sorting offered us a way of doing so through simply counting the number of elements in the list. This enables a straightforward configuration of a training curriculum.

For training the modules independently, we rely on programmatically generated execution traces. In order to constrain the problem domain and rule out possible representation issues, we only consider the domain of lists containing integers from 0 to 9. We also take the *Selection Sort* algorithm as reference algorithm. Despite its $\mathcal{O}(n^2)$ time complexity, *Selection Sort* is among the simplest sorting algorithms.

We let the problem state be represented in an environment where the agent must rely on two pointers (A and B) for traversing the list. We therefore expect the problem to be solvable through the use of five primitive operations, which we will train for carrying out the correspondingly intended sub-tasks.

- `mova`. Moves the pointer A one position to the right.
- `movb`. Moves the pointer B one position to the right.
- `retb`. Returns the pointer B to the position right after the pointer A.
- `swap`. Exchanges the values located at the positions pointed by A and B.
- `EOP`. Leaves the representation unchanged and marks the end of execution of the perception-action loop.

Each problem instance can be solved using this set of primitive operations. In the initial configuration of the environment, pointers A and B are placed at the first and second positions respectively. We define final states as those having both pointers at the last position. In any case, the execution may stop when the agent selects the `EOP` operator.

In our implementation, every integer in the list is represented as a one-hot vector. Pointers are represented as one-hot vectors too, although along the sequential dimension i.e. the sequence is all zeros except where the pointer is

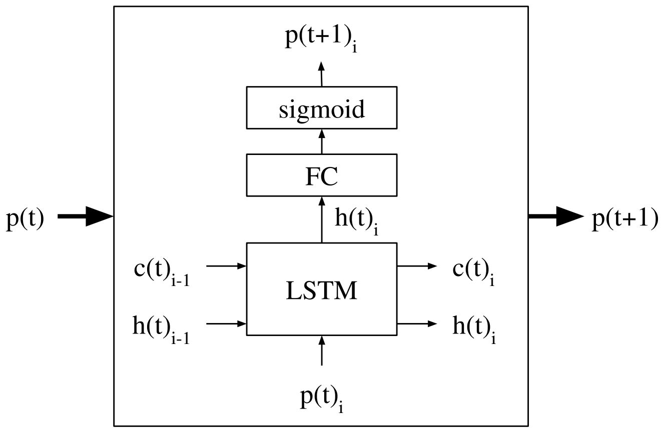


FIGURE 3.3: Architecture of the pointers' functional submodule, with an LSTM and a fully connected output layer with sigmoid activation. $c(t)_i$ and $h(t)_i$ are the LSTM's internal state and output at each time step and position i.

located.

Given the sequential structure of the data, functional modules are implemented by RNNs. Specifically, we let every functional module be implemented by LSTMs with 100 hidden units. The `swap` module relies on a bi-directional LSTM with a final softmax layer (Figure 3.4) and for those modules that modify the pointer we just apply a sigmoid activation, thus not enforcing the one-hot constraint (Figure 3.4).

The control module is composed by the *digestor* and the *controller* (Figure 3.5). The *digestor* is based on an LSTM and generates a fixed-size embedding of the environment representation, which is then fed to the *controller* every time step. The *controller* is therefore learning the actual control policy, which decides what operator to apply next conditioned on previous actions and states.

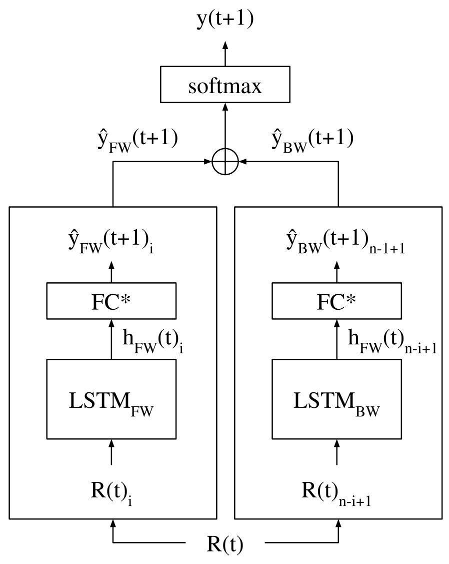


FIGURE 3.4: Architecture of the `swap` functional module. The entire representation is merged into a one single tensor and fed into a bidirectional LSTM. The outputs pass through a fully connected layer and are then merged by addition. *Fully connected layers share parameters.

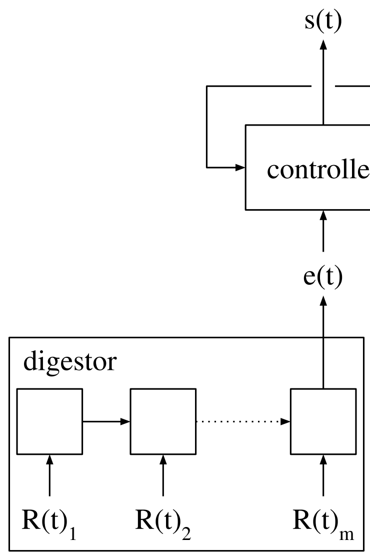


FIGURE 3.5: The *digestor* creates a fixed-size embedding $e(t)$ from the state representation and the *controller* takes it as input at every execution step. Conditioned on the embedding and its past state, it outputs the selection vector $s(t)$.

3.2.2 Training considerations

At this point we started seeing some differences between the monolithic and the modular training approach. We found that monolithic end-to-end training is not possible unless we help it by using target traces, which contribute to an auxiliary selection loss and override the operator selection during training. This method has been previously used in neural program synthesis with success (Bunel et al., 2018).

On the other hand, modular training forces us to find ways to prepare modules for compositional execution. For that matter, we rely on the addition of noise to the input and measuring output saturation. Our assuption is that, if the noise present in the output is less than the maximum expected input noise, then modules present self-correction abilities and can be chained together while keeping functional stability. This kind of assumptions is not new and has been successfully relied on in the past for achieving compositional behaviour and facilitate planning in robotics (Burridge, Rizzi, and Koditschek, 1999).

From these observations stem two measures that we use for characterizing the state of training. We measure first the error rate by comparing the output with the expected result. Additionally, pointers can make an output invalid if they are not consistent (only one value in the sequence is above 0.5). Secondly, we measure the saturation of the outputs by counting the percentage of values that differ in more than 0.1 with respect to the one-hot references.

The monolithic configuration is trained until the error rate is 1% or less. Additionally, operation modules require the saturation error to be below 1% too. We may also stop training if the training loss becomes stagnant or falls below 10^{-6}.

We add noise to modules' inputs following equation 3.1 and extend the output vocabulary for allowing the generation of null vectors. Noisy one-hot vectors may be passed through a softmax operation in order to meet the constraints of the softmax manifold (see eq. 3.2).

$$\hat{x}_{uniform} = |x - U(0, 0.4)| \tag{3.1}$$

$$\hat{x}_{softmax} = \text{softmax}(\hat{x}_{uniform} \cdot 100) \tag{3.2}$$

We train all configurations under full supervision, with cross-entropy loss and Adam (Kingma and Ba, 2014) with learning rate 0.001. Results for the monolithic configuration are averaged over five runs.

3.2.3 Comparison of training times and complexity

Despite having to comply with stronger output criteria, the training time is around one order of magnitude lower for the modular configuration (Figure 3.6). This graph assumes that modules are trained sequentially, but the training time can be reduced even further if all modules are trained in parallel.

We show in Figure 3.7 how each training progresses in a very different manner. The modular configuration needs more training steps but much less time than the monolithic one. Modular error rates are also less variable. In Figure 3.8 we show that the gradient is much richer in the monolithic case, with a higher mean absolute value per parameter and greater variations, which is a possible explanation for the higher per-step training efficiency.

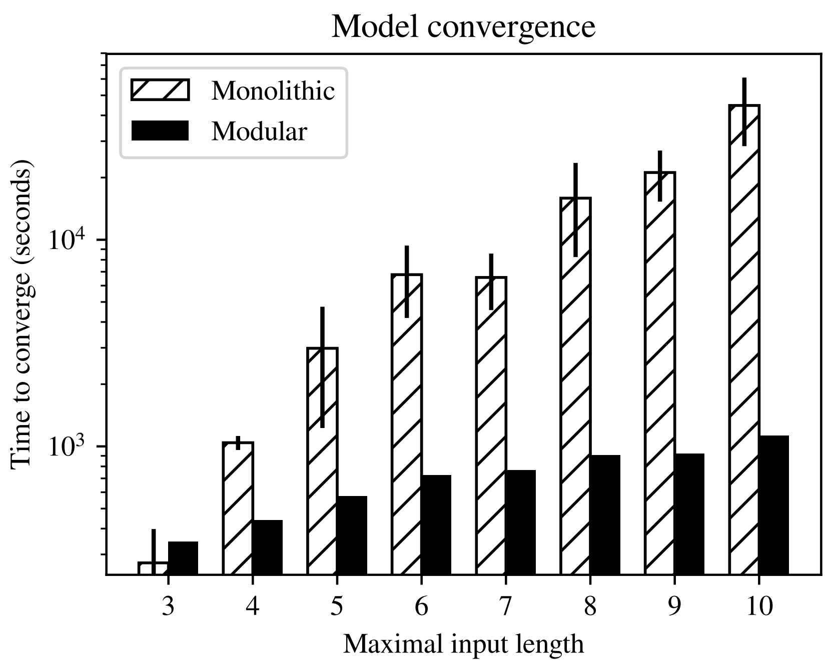
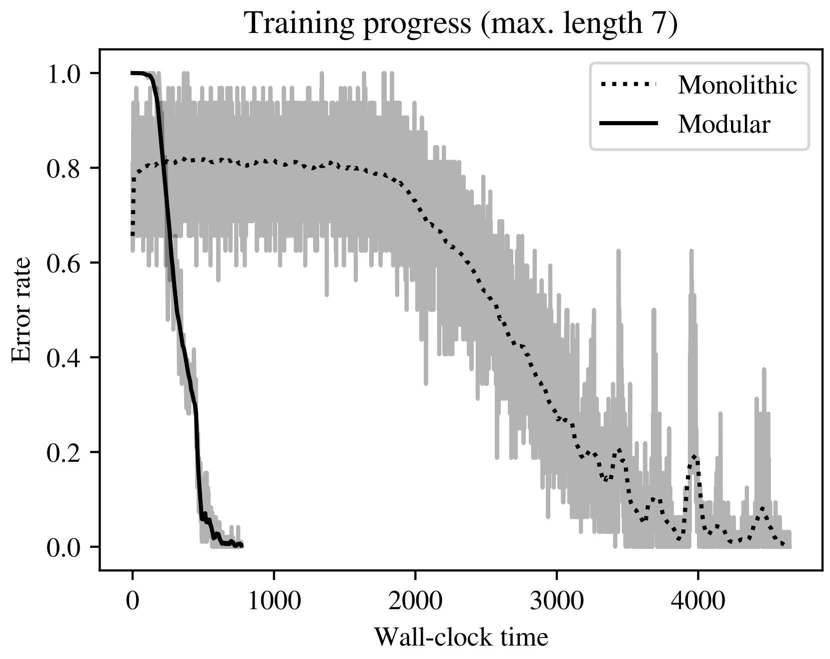


FIGURE 3.6: Convergence times for modular and monolithic configurations, represented in logarithmic scale.

In Figure 3.9 we plot the training loss for operation modules during independent training. Pointer operations are functionally simple and converge very quickly, learning to delay the input in one time step. The swap operation is also relatively simple, only needing to remember one digit (see example in Listing 3.1), although the loss landscape appears to be quite flat after the fast initial convergence. The control module starts at higher loss values, but converges sooner despite having to digest the list into an embedded representation and condition its output on past actions. Evidence indicates that this might be also caused by a richer gradient, product of the sequential unrolling of the module. In absence of significant gradient explosion or gradient vanishing, the variations in the gradient greatly inform the optimization.



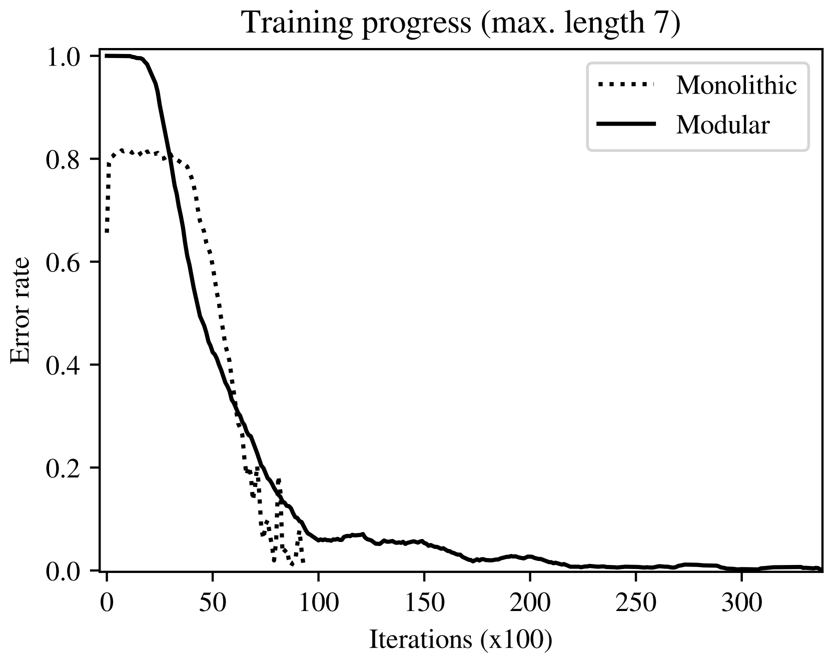


FIGURE 3.7: Error rates during training for modular and monolithic configurations, with respect to time (top) and training iterations (bottom).

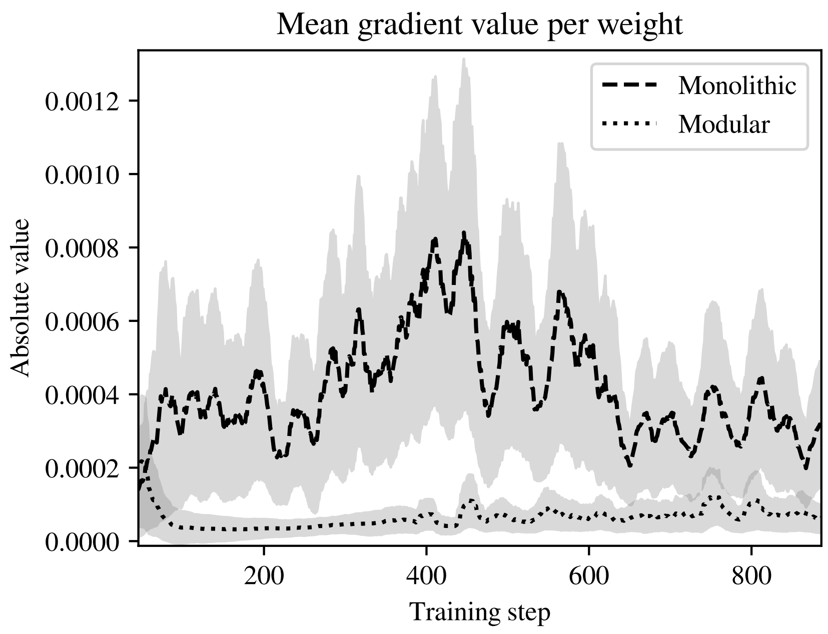


FIGURE 3.8: Mean absolute value of the gradient at the weight level for each configuration. This data was obtained while training with lists of 7 digits maximum. Obtained data was slightly smoothed to help visualization. The grey shadow represents the standard deviation.

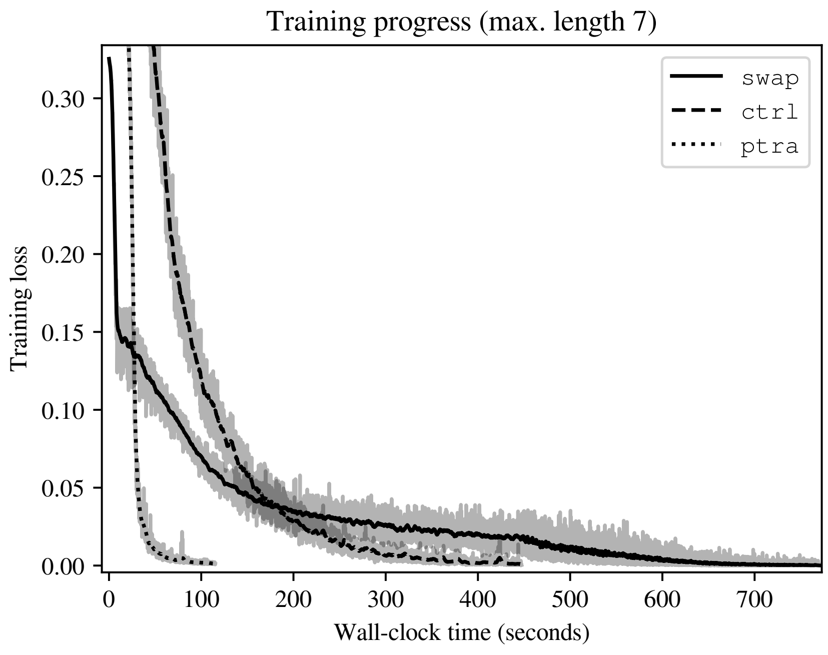


FIGURE 3.9: Progress of the training loss for the different operations during training with lists of maximal length 7. We only show `ptra` as the complexity is identical for `ptrb` and `retb`.

LISTING 3.1: Example of a possible implementation of the `swap` operation carried out by a bidirectional LSTM. Underscores represent zero-vectors.

```
List      = 3,9,5,4,_
Pointer A = 0,1,0,0,0
Pointer B = 0,0,0,1,0

Forward  LSTM Output = 3,_,5,9,_
Backward LSTM Output = _,4,_,_,_
Merge by addition ---> 3,4,5,9,_
```

3.2.4 Generalization over longer sequences

When it comes to modular NNs and especially to training its modules independently, one common concern is the level of compositional generalization that the whole ensemble will exhibit after training. This is a form of generalization different than the typical one, in which the main goal is to see how much of the performance is lost when unseen samples are presented to the network. In this case, we are interested in seeing how the model behaves when the number of operations that are chained together differs from training and, in the modular configuration, whether the individual training and the de-noising augmentation (Equation 3.1) are effective for that matter. Because we want to exclude any generalization factor unrelated to the composition of modules, we only show the model integer numbers in the range [0,9], but we include in the test set lists that are longer than those seen during training.

In Figure 3.10 we show the generalization of each configuration on lists

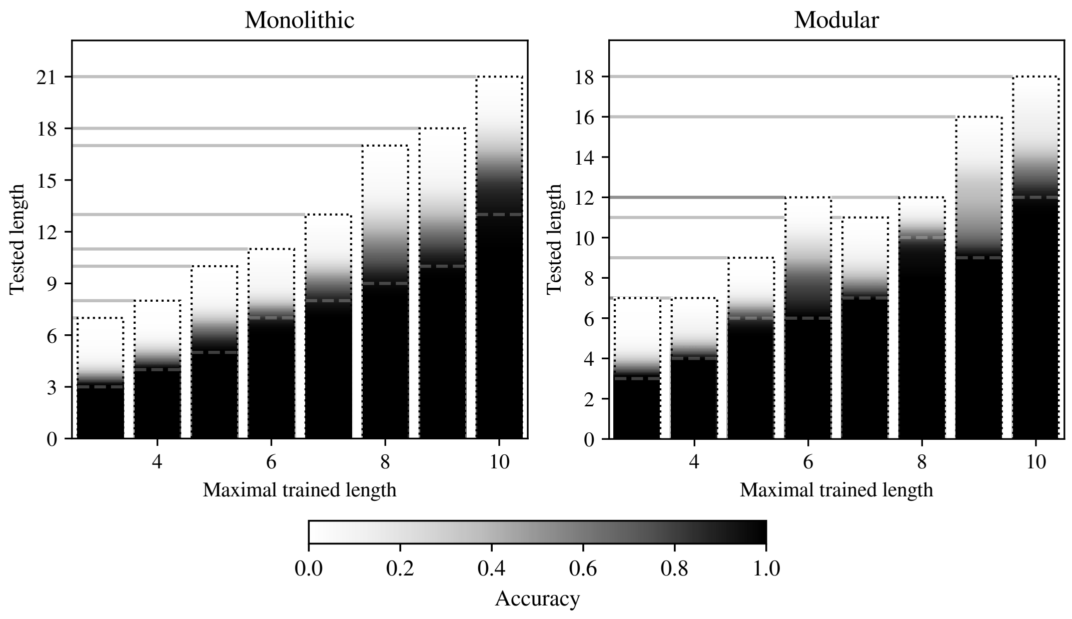


FIGURE 3.10: Generalization tests for monolithic (left) and modular (right) configurations. Horizontal black lines mark the length where 0 accuracy is achieved. A dashed line on the bar indicates where the accuracy passes 0.9.

with lengths unseen during training. Contrary to our initial expectations, the monolithic configuration generalizes better to longer lists and its performance degrades slower and smoother than on its modular counterpart. We hypothesize this could be a side effect of the training noise and saturation requirements. This phenomenon must be therefore taken into account when designing modular NNs.

3.3 Main results

In this initial approach to modular-designed NNs based on a perception-action loop, the whole system is functionally divided in operation modules with standardized interfaces. We have shown that modularity has a very positive impact on training speed and stability. In terms of compositional generalization, we have introduced noise in the inputs with the intention of embedding noise-reduction capabilities into the modules and therefore making them work with one-hot vectors, as well as with non-saturated inputs. However, this mechanism ended up increasing the training difficulty and also the method failed to induce the intended behaviour. In consequence, we observed irregular generalization to longer sequences after this particular configuration of independent modular training.

The results indicate that the reliance on modular NNs definitely leads to a better utilization of computational resources and data available, as well as an easier integration of expert knowledge. Thanks to the incorporation of interfaces, which isolate implementation from particular use-cases, operation modules can be easily upgraded or switched by alternative implementations. We have also demonstrated how independent modular training can cope with high degrees of recurrence, which otherwise pose a significant challenge to monolithic end-to-end optimization methods.

Chapter 4

Improving reusability and compositionality of neural modules

Having gathered knowledge about the low-level phenomena of modular training (see §3), we are now more prepared to jump up a level of abstraction and focus on the main issues observed: compositionality and therefore generalization to different modular layouts. In this chapter we will go over the different compositional dependencies that may happen, analyzing them and proposing ways of going around them during modular training. We then propose a methodology for the design of composable modular interfaces and, in the process, present a couple of novel concepts and design and training tools, as the *Learning by Role* phenomenon and the Surrogate Gradient Module.

We exemplify the complete method on a case study based on Neural Module Networks (Andreas et al., 2016, NMN), which is oriented to a VQA (Antol et al., 2015) task of significant complexity. Additionally, we do compositionality tests on the CLEVR dataset (Johnson et al., 2017), a synthetic VQA task that removes most of the bias present in natural language. The synthetic nature of this dataset allows us to have access to execution traces, this way setting the complexity introduced by natural language aside to center the comparison on the compositional part of the task.

4.1 A taxonomy of compositional modular dependencies

Independent training of neural modules usually leads to different logistic issues characterized by the difficult access to first-hand training input data and targets. Paying close attention to these cases we have identified five main scenarios:

- **Decoupled input**. The input data needed for training the module is available right away from a data source. This is often the case of input modules or modules representing the first operations in the pipeline.
- **Dependent input**. The module's input is another module's output and cannot be obtained by other means than executing the latter. This sequential dependency forces the corresponding input modules to be trained beforehand.
- **Direct supervision**. There is a loss function that can be computed directly over the module's output and allows training it in a fully-supervised manner while targeting the intended functionality (i.e. not just an auxiliary loss).
- **Indirect supervision**. There is no loss function that can be applied directly on the module's output for training it. A suitable gradient might be computed via backpropagation through other modules.

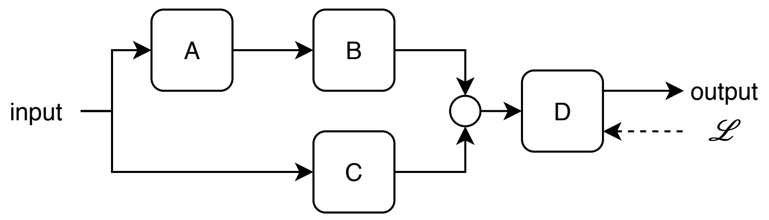


FIGURE 4.1: Example arrangement of four neural modules. Modules A and C receive decoupled input and only D has direct supervision. Module B has neither one nor the other. Outputs of B and C may interact before going into D, exhibiting codependent gradient.

- **Codependent gradient**. The module's output is combined with other elements in a way that affects the gradient's direction (i.e. in any other way that is not concatenation, addition or averaging).

We exemplify all cases enumerated above with a hypothetical static assembly of four neural modules (Figure 4.1). We assume a hypothetical task that is carried out by this assembly, for which there exists a dataset that links global inputs to targets for the final output. Among the modules that implement the corresponding sub-tasks we may identify modules with decoupled input (A, C), dependent input (B, D), direct supervision (D) and indirect supervision (B, C). Additionally, and depending on the way the outputs of modules B and C are merged together, these modules might present a case of codependent gradient.

Straightforward to handle are the cases of decoupled input and direct supervision. Upon decoupled input one can just make use of the already available input data present in the dataset, and direct supervision ensures the availability of a loss function for conducting fully-supervised training. In the following we will propose some guidelines and methods for solving the other cases or at least minimizing their associated adverse effects.

Modules with dependent input introduce sequential dependencies in the training process. However, our goal is to be able to train these modules independently and without requiring any sort of retraining or fine-tuning of other modules, regardless of their placement downwards or upwards in the pipeline. In order to accomplish this, we propose the creation of auxiliary datasets, which store the relationship between some module's outputs and the corresponding nearest targets available for supervised training. These datasets are intended to remain constant as long as modules' implementations do not change and therefore provide a data-wise abstraction for modular training. With the help of some additionally stored data, like original sample IDs, a certain degree of coordination can be accomplished in order to train modules that rely on the output of several other modules.

Cases of indirect supervision require finding a substitute gradient for training the module. During monolithic end-to-end training, these cases rely on the highly complex chain of modules that generate the final output and the backpropagation algorithm. In the absence of such modules, we propose the use of an alternative auxiliary module that is capable of delivering the required learning signal whilst enhancing the module's compositional properties (§4.2.3).

Lastly, cases of output interdependence might not be obvious to identify but they do have severe implications regarding modular training and they are not trivial to solve. The consequences of these cases only show up during the backward pass and that is where the difficulty resides. If two modules' outputs are merged like $\alpha \cdot x + \beta \cdot y$ that is totally fine, as the gradients will take the form of $\alpha \cdot \nabla_x$ and $\beta \cdot \nabla_y$. The gradient will therefore be affected solely by a scalar factor, which is equivalent to changing the learning rate, but it is no real issue nevertheless. However, something as innocent as $x \cdot y$ will make the gradients be $\nabla_x \cdot y$ and $x \cdot \nabla_y$, introducing so a rather inconvenient interdependence during the backward pass. This latter case would probably require joint training of the whole set of dependent modules, but somewhere between the first example and the second one is where sequential training can be leveraged (§4.3.2).

4.2 Compositional modular interfaces

In the context of software engineering, interfaces are intended to foster the reuse of functionalities by decoupling the input and output representation from the internal implementation. This in turn eases scaling and maintenance of the system. The very same concept can be extrapolated to NNs in order to further reduce inter-modular coupling and boost compositionality and generalization to novel layouts, but NNs introduce some new aspects that must be taken into consideration. Especially, the possibility of NNs overfitting to arbitrary features of the input distribution (Novak et al., 2018) can give raise to inter-modular coupling, even if not having been trained jointly.

The architectural framework presented in section 3.1 already proposes the use of interfaces in neural modules, which set an abstraction layer between the representation of the environment used by the agent and the representation used or required by the internal implementation of the module. Merely by having this abstraction interface, the possibilities for functional reuse, maintenance or complete re-implementation of its functionality increase greatly. However, and as we showed in section 3.2.4, that abstraction is not enough in terms of avoiding inter-modular coupling and favouring compositionality through combinatorial extrapolation.

4.2.1 Neural data types

Our end goal is to make modular functionality totally independent from the implementation and make it follow closely the intended specification. In order to do that, we must stay away from non-typed latent representations (unless the specification explicitly requires so) and rely instead on typed representations that are consistent with a generic interpretation of the task and facilitate its generalization. According to the universal approximation theorem (Lu et al., 2017; Hanin and Sellke, 2017), NNs can approximate functions with arbitrary precision, which depends on the number of hidden units. However, this theorem was presented as only being valid if the input has *support in the unit hypercube* (Cybenko, 1989), therefore providing no guarantee when it comes to extrapolating outside the range of values present in the training data. For this reason we encourage the use of bounded input values, which favour the interpolation regime and avoid off-distribution issues (Gleave et al., 2020). This principle is very well exemplified in Cai, Shin, and Song (2017), being the first case of guaranteed generalization in NNs.

We can find several examples of some of these features in the literature, mostly disguised as obvious design choices —like gating units (Hochreiter and Schmidhuber, 1997)— or convenient representations —e.g. latent variables in generative adversarial networks [see Appendix E in Brock, Donahue, and Simonyan (2019)]. Softmax layers enforce manifold constraints over the set of output vectors, ensuring that all values are in the range $[0, 1]$ and that they all add up to one.

We propose to intentionally impose output constraints, so that the output distribution can be matched with some target distribution associated with the intended data type. In order to do that, there are two types of constraints to apply:

- Hard constraints set the domain of the output values via clipping or activation functions. All output values will therefore remain in the domain by definition.

- Semantic or soft constraints determine in detail the expected manifold for the output tensor. Sometimes they can be implemented as afterward filters (e.g. softmax or normalization), but quite often semantic constraints must be implemented as a form of training loss, which may be defined through labeled data, heuristic functions, a discriminator or any other kind of auxiliary training module (see section 4.2.3).

4.2.2 Learning by Role

Learning by Role is a side-effect of end-to-end optimization, a phenomenon by which a part of a NN acquires a functionality that best fits the needs of the whole architecture in order to solve the task. This phenomenon is strongly related to meta-learning (Flennerhag et al., 2021), where an inner-loop function is optimized by means of an outer-loop loss (meta-loss). In meta-learning, however, no loss function is specified directly on the inner-loop part, so the inner-loop optimization is a byproduct of the outer-loop optimization. The *Learning by Role* phenomenon is named after an equivalent technique from the field of language learning (Ladousse and Maley, 1987, *Role Play*), where students develop a set of skills in a simulated scenario that they can later on leverage in the real world.

Nevertheless, *Learning by Role* is a phenomenon that takes place in any kind of supervision setting, as long as one part of the network is subject to another part of the same network and training is conducted in an end-to-end fashion. Andreas et al. (2016) described it as when a part of the NN *acquires its behavior as a byproduct of the end-to-end training procedure*.

According to *Learning by Role*, each part of the network learns to do its best within the realm of scenarios it is allowed to express itself in. We can find many examples of this phenomenon in the literature, from the hierarchy of features extracted by CNNs (Yosinski et al., 2015) as an implicit requirement of an image-classification task, to the learning of very specific manifolds through adversarial play (Goodfellow et al., 2014). Especially relevant for us is the case of the NMN architecture, in which modules are devised to perform a concrete task and then they learn that specific target functionality as a consequence of their placing in the different modular layouts (Andreas et al., 2016; Hu et al., 2017). Allegedly it is the correlation or coherence in the gradients that a module receives in such situations what makes it consistently converge to the expected behaviour.

4.2.3 The surrogate gradient module

In an end-to-end training scenario, when a module provides input to a complex NN, the whole chain of operations is responsible for the final loss and therefore also for the gradient that is back-propagated to the module. This dependency shapes in turn the behaviour of that initial module, a phenomenon that we call *Learning by Role* (§4.2.2). Although one might think that a module trained under these conditions cannot be trained in other way, and that the whole upper part of the NN is required for that matter, during training, gradients are used mainly as a directional resource, which means that the quality of gradient can be somewhat

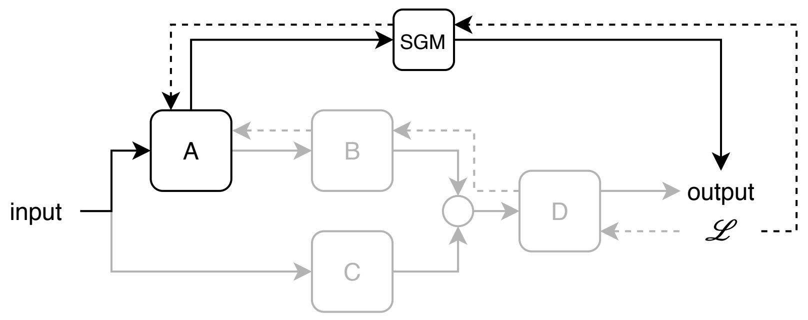


FIGURE 4.2: SGM bridging the gap between module A and the loss function, substituting this way the gradient provided by modules B and C. Solid lines represent data flowing forward and dashed lines represent gradients being back-propagated.

overlooked as long as it provides a good approximation of the weight update direction (Bernstein et al., 2018; Mordido, Van Keirsbilck, and Keller, 2020). As a matter of fact, SGD not only measures the gradient at one single point of the weights manifold, but also approximates it from an extremely small subset of the data. It is however the correlations and trend of the different steps taken what makes the weights progress —on average— in the right direction.

Based on these arguments, our hypothesis is that we can train a neural module without direct supervision, by handcrafting instead an auxiliary module that implements all requirements that the module's output must comply with. The auxiliary module can be in principle very simple in complexity, just like learning the rules of a game is a much easier task than learning to master the game in question. We call this auxiliary module the *Surrogate Gradient Module* (SGM), since it is intended to replace the original gradient during training, and we place it right between the module's output and the nearest loss function, connecting both of them (Figure 4.2).

It is essential to keep in mind that the SGM is not a surrogate model of the missing functionality nor it intends to be such thing. Actually, the performance during inference (classification or regression) can be arbitrarily bad, as long as it does not negatively impact the average gradient direction. The SGM is a surrogate model of the gradient previously provided by that missing functional block. This is a key distinction because the training gradient can be modelled in much simpler terms, thus surrogating the training procedure to a much simpler chain of operations.

As a final explanatory resource, let us compare two different training setups. In both cases, the NN is composed of two modules A (input) and B (output), but we are interested in module A as the final product of the training procedure. Module A is the same in both scenarios, but module B has two versions: B_1 is highly complex and B_2 is relatively simple. It is therefore possible that, following Equation 4.1, both optimization procedures arrive to similar solutions for the module A.

$$\exists f \neq g : \arg\min_{\theta} \ell\left(f \circ h(x;\theta), y\right) \approx \arg\min_{\theta} \ell\left(g \circ h(x;\theta), y\right) \tag{4.1}$$

Design considerations

Despite our expectations regarding the task-level performance of the SGM, we hypothesize that we can leverage it for having a relative improvement indicator (§4.3.1). However, uncertainty measurements must be considered too, for an untrained module might hit good metrics by chance. Additionally, the SGM itself is usually parameterized and these parameters might need some time to adjust before giving any sensible indication of relative performance. In any case, we strongly recommend to follow the design considerations below:

- A transformation from the module's output to the supervised data type must take place.
- The transformation must be consistent with the known relations existing between the module's output and the prediction target. Arbitrary data conversions will probably result in some form of overfitting.
- All functional features of the input module must be involved in the transformation, thus forcing the module's output to comply with the same constraints of the full task.
- The SGM must be fully differentiable in order to allow the gradient information to get through and this way carry the implemented training biases.

In general, one question worth stating while designing a SGM is: can the trained module minimize the loss function while not performing the targeted function? If the answer is no, the design of the SGM is complete. If the answer is yes, there are still constraints left to be implemented. We provide an example of how a SGM can be built by following these guidelines in the upcoming experimental section.

4.3 Experiments

We conduct a series of experiments, focused on testing the effects of applying the modular methodology proposed in the previous sections to an already existing modular network. We take the first implementation of NMN as a model reference.

Neural Module Networks (NMN) is a class of neural architectures introduced by Andreas et al. (2016) in which the NN is formed by a set of composable neural modules that are assembled differently, depending on the input sample (Figure 4.3). Modules have different architectures depending on their expected functionality, and they also adopt different functionalities as a consequence of the end-to-end training (§4.2.2).

In the NMN architecture, the Stanford Parser (Klein and Manning, 2003) takes the question as input and informs the construction of the modular layout. A VGG16 convolutional neural network (Simonyan and Zisserman, 2014) pre-trained on ImageNet (Krizhevsky, Sutskever, and Hinton, 2012) extracts image features and an LSTM (Hochreiter and Schmidhuber, 1997) maps the input question to a distribution over possible answers. We show in Figure 4.4 a visual representation of all functional module's architectures.

We first compare end-to-end training versus modular training on two setups in the VQAv1 task. In one case we leave all modular interfaces untouched and in the other case we conduct an analysis and re-design of modular interfaces, implementing also neural data types as described in section 4.2.1. We only use the training split of the VQAv1 data set during training and we test both on the validation and official test splits. Secondly, we perform compositionality tests on the CLEVR data set. The code for the experiments in this section can be found at `https://github.com/dcasbol/dnmn`.

In the NMN architecture, the functional module `Find` is always used upfront in the pipeline, with the purpose of extracting visual maps that will be later used by other modules. It is a filtering and feature-extraction module, and under no circumstance it is given any direct training loss. However, we can still leverage a SGM for training the module independently from the others.

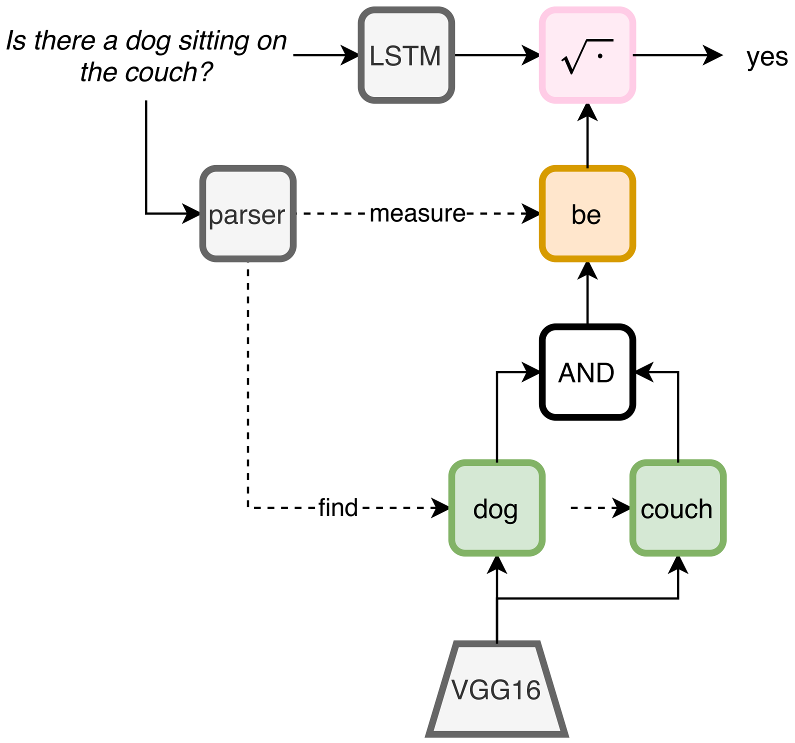


FIGURE 4.3: Original NMN architecture. A parser determines the module layout from the input sentence. An LSTM processes the sentence separately and both answers are combined via a geometric average.

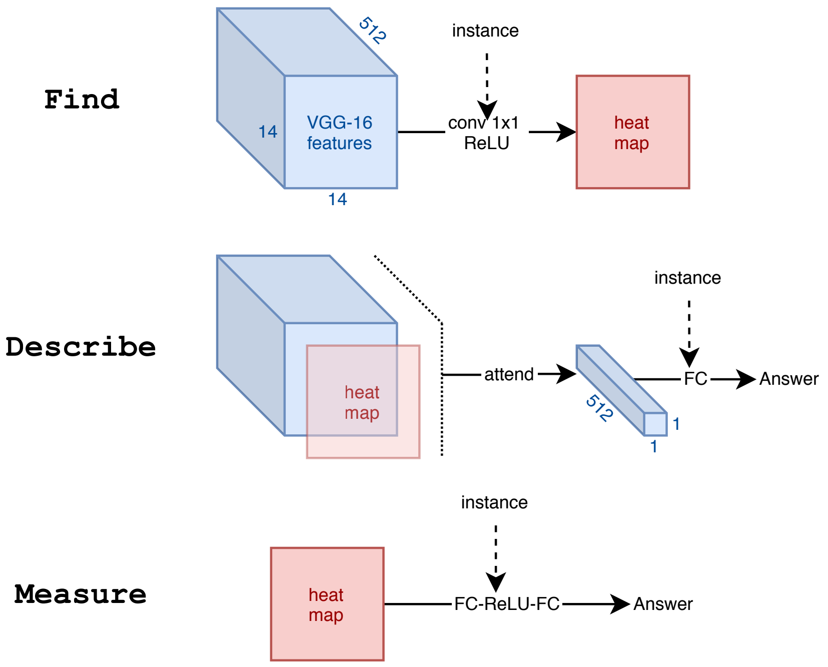


FIGURE 4.4: Schematic depiction of NMN's composable modules. The module's weights are distinct for each instance and the attention mechanism is based on multiplication and weighted average.

Keeping the restrictions over the module's output is extremely important for ensuring that the gradient guides the training to an equivalent configuration. For that purpose, our implementation of the SGM incorporates the attention mechanism and takes the attended features or directly the module's output depending on the question type (see Figure 4.5).

Additionally, and serving as an empirical proof that gradient approximation does not require a high-fidelity approximation of the forward function (see §4.2.3), we impose a low-complexity constraint on the SGM by factorizing the auxiliary module's weight matrix in two smaller matrices. Let $W \in \mathbb{R}^{N\times M}$ be the weight matrix of the SGM, we build it as a multiplication of $W_A \in \mathbb{R}^{N\times L}$ and $W_B \in \mathbb{R}^{L\times M}$, where $L \ll \min(N, M)$.

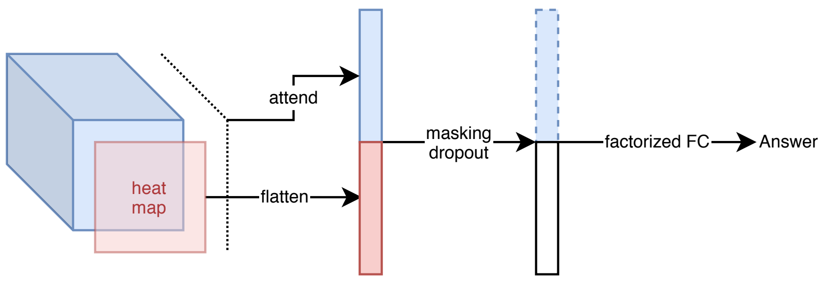


FIGURE 4.5: SGM used to train the `Find` module. The input is processed in an equivalent way as in the full NMN, although with a single factorized linear matrix. Dropout is leveraged for computing predictive uncertainty.

4.3.1 Validating the relative-improvement hypothesis

In section 4.2.3 we hypothesize that the classification performance of the SGM, despite being bad in absolute terms, serves as a relative indicator of the trained module's performance. If this is true, any performance measure taken over the SGM's output should correlate with that of the complete architecture, if both instances share the weights of the module in question.

In order to test this hypothesis, we first train via surrogate gradient a set of `Find` modules with different expected performances by assigning each of them a different set of hyperparameters and number of training epochs. We then record their respective SGM's losses and predictive variances on a separate split of the training data. We finally rely on this predictive variance for selecting the modules closest to the loss-variance Paretto front (see Figure 4.6).

As a last step, we transfer the weights of each selected `Find` module to the complete architecture and let it train end-to-end until convergence, all while keeping the weights of the `Find` module frozen. Results in Figure 4.7 show a correlation between surrogate losses and losses from the entire architecture, which has a Pearson correlation coefficient of 0.9 with a p-value of 10^{-14}. This proves

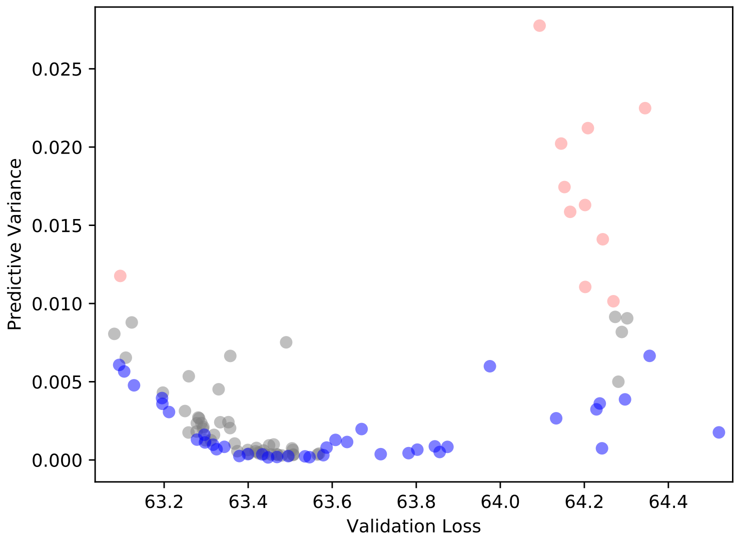


FIGURE 4.6: Validation losses and predictive variances of independently-trained `Find` modules. Selected modules are shown in blue.

therefore that the SGM is a useful tool not only for training modules in isolation, but also for conducting hyperparameter optimization or early stopping. It is also worth noting that some degree of noise is to be expected in the measurements, most probably due to the effects of random batching and under-training.

4.3.2 Ablation study of neural data types

Training a NN in a modular fashion involves three main elements, namely that the architecture is modular, that inter-modular dependencies are taken care of and that compositional interfaces with neural data types are implemented. In this experimental section we will apply the proposed guidelines in order to train the NMN architecture in a modular-wise way, and we will do so in two stages with the aim of showing the impact of neural data types on the overall performance of the network. Additionally, we will point out several other aspects which

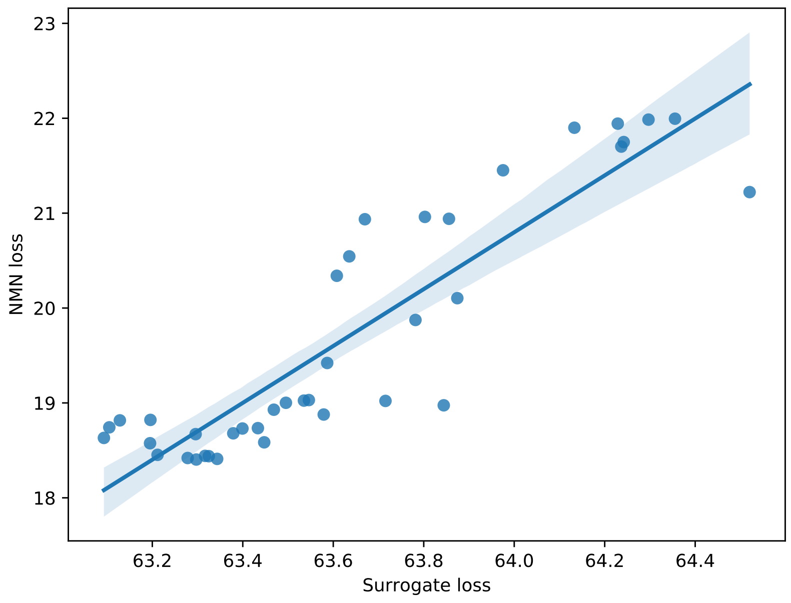


Figure 4.7: Correlation between SGM losses and losses on the complete architecture. The line shows the correlation found and the shadowed area represents the 95% confidence interval for the regression line.

make modular networks and modular-wise training an attractive solution towards optimizing computational resources.

Solving inter-modular dependencies

One first shallow approach to applying the methodology might be to only analyze and resolve the dependencies described in section 4.1. We show in Table 4.1 the set of compositional dependencies found in the NMN architecture, which we will then proceed to solve. In the following, we assume that all modules must be trained and that there is no module available that we can reuse or re-purpose. We encourage the reader to keep in mind that this scenario is an extreme case and a worst-case scenario for independent modular training, which often benefits from the existence of pre-trained modules and parallel training.

Module	Input		Supervision	
	Decoupled	Dependent	Direct	Indirect
`Encoder`	✓		✓	
`Find`	✓			✓
`Describe`		✓	✓	
`Measure`		✓	✓	

TABLE 4.1: Compositional dependencies found in the NMN architecture.

From Table 4.1 we quickly identify that the `QuestionEncoder` can be trained right away because of the availability of decoupled input and direct supervision. Additionally, we also observe that an SGM is needed for training the module `Find`, as it is the only module that receives gradient indirectly, through other modules. Present in the table but also from looking at the architecture layout (Figure 4.3), we see that modules `Describe` and `Measure` depend on the output of module `Find`, and must therefore be put on hold until the latter is ready before it can be trained.

We alleviate the sequential dependency of modules `Describe` and `Measure` via the creation of an intermediate dataset, in which we record the heat maps and attended vectors that resulted from applying the module `Find` over the original VQA samples. This not only allows training these root modules more efficiently, without running the `Find` module, but also enables parallel modular re-training in the future.

We optimize hyperparameters independently for each module and select the configurations that perform best over a separate validation split. We then put all modules together and test the full assembly on the test set, achieving an accuracy of 54.49%, which closely matches the performance of the end-to-end trained baseline (see Table 4.2).

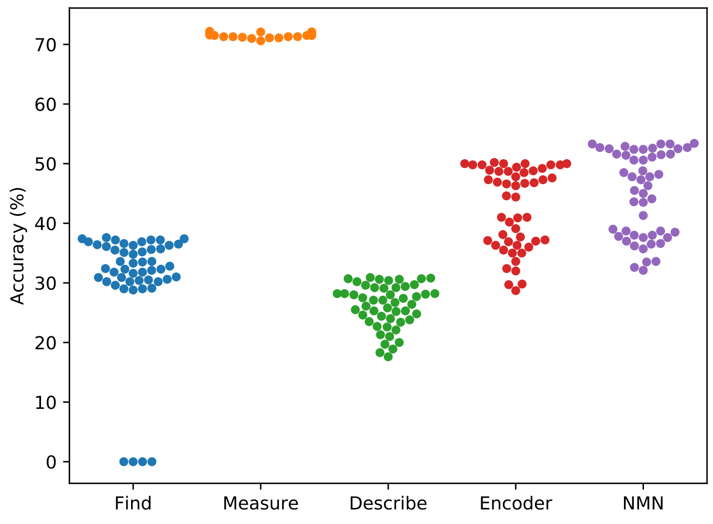


FIGURE 4.8: Accuracy obtained for every modular and end-to-end evaluation during hyperparameter optimization.

Independent modular training also allows for a more detailed inspection and analysis of modular training and performance. In Figure 4.8 we show the performance achieved by all hyperparameter configurations tried on each module and also during the end-to-end hyperparameter optimization. This makes it possible, among other things, to identify the degree of susceptibility to hyperparameters of each module or to determine which module is worth putting more resources into in order to improve the overall performance.

As an example, we can compare the performance distributions of modules `Measure` and `Describe`. The first is very concentrated, while the latter shows more dispersion. Knowing this, and also that the `Describe` module is used on 64.19% of the training questions versus 35.81% in the case of the `Measure` module, we could want to focus resources on improving the module `Describe` and that way making the most out of our computational investment.

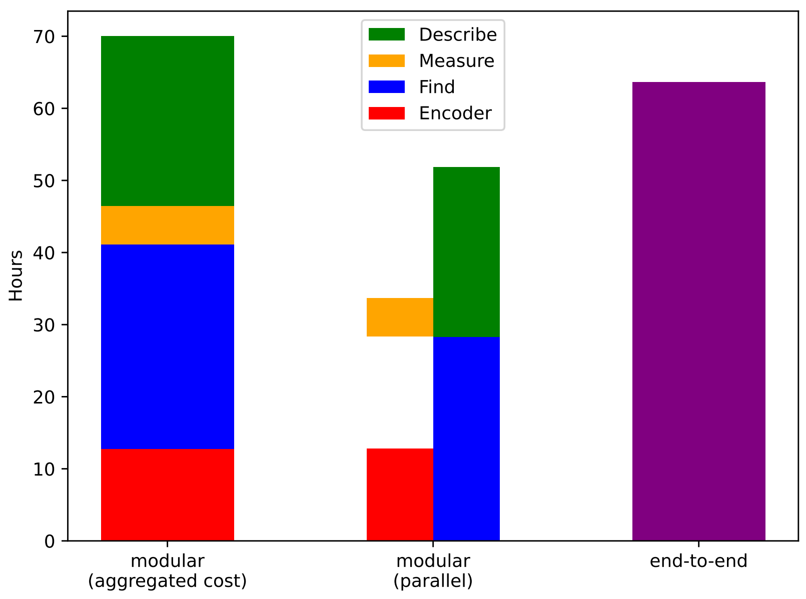


FIGURE 4.9: Total training times including hyperparameter optimization. The cost for the modular approach is commonly a fraction of the aggregated costs, depending on the availability of pre-trained modules and the number of modules trained in parallel.

We show in Figure 4.9 the total cost in training hours of the hyperparameter optimization process. The time invested in generating virtual dataset did not exceeded five minutes and is therefore not represented in the figure. We observe that training any module in isolation represents only a fraction of the cost of training the full architecture end-to-end. Aggregated costs of sequentially training all modules add up to a 10% increase in training costs, but this scenario is quite unrealistic since it does non consider the availability of pre-trained modules and disregards the possibility of training a subset of the modules in parallel (i.e. first `QuestionEncoder` and `Find`, then `Measure` and `Describe`, Figure 4.9 middle). Merely considering parallel training would bring the cost of independent modular training down to roughly an 80% of the end-to-end training cost.

Applying neural data types

In the previous section we showed how modules could be trained independently without hurting performance and actually obtaining some benefits in terms of training time and network analysis and maintenance. However, modular interfaces were kept the same, so the compositional behaviour of the modules was left untouched or even slightly damaged, due to the high sensibility of non-typed interfaces to changes in modular composition (especially when chaining many modules one after another, which heavily scales values up and down). We exemplify in this section how to implement interfaces with data types (§4.2.1) that stand the test of modular composition.

Setting the focus back on the NMN architecture, Andreas et al. (2016) introduce the `QuestionEncoder` module with the intention of modelling syntactic and semantic regularities present in the data, mentioning also the goal of giving the network some sort of commong sense. The original implementation merges the softmax outputs of the `QuestionEncoder` and the root module via a geometric average, which is problematic for two reasons: first, it allows masking out the other module's answer, tying the answers to the biases present in the data and preventing new contradicting evidence to override it (e.g. detecting a green dog in the picture and not being able to answer 'green'); second, although related to the first point, it forces the modules to learn functions of higher complexity than needed, having to give non-zero probabilities to some answers if they are even remotely possible according to the training data.

We instead interpret the functionality of the `QuestionEncoder` module as providing some prior about the possible answers. This prior gives some answers a higher probability than others but it does not rule out any other answer if evidence

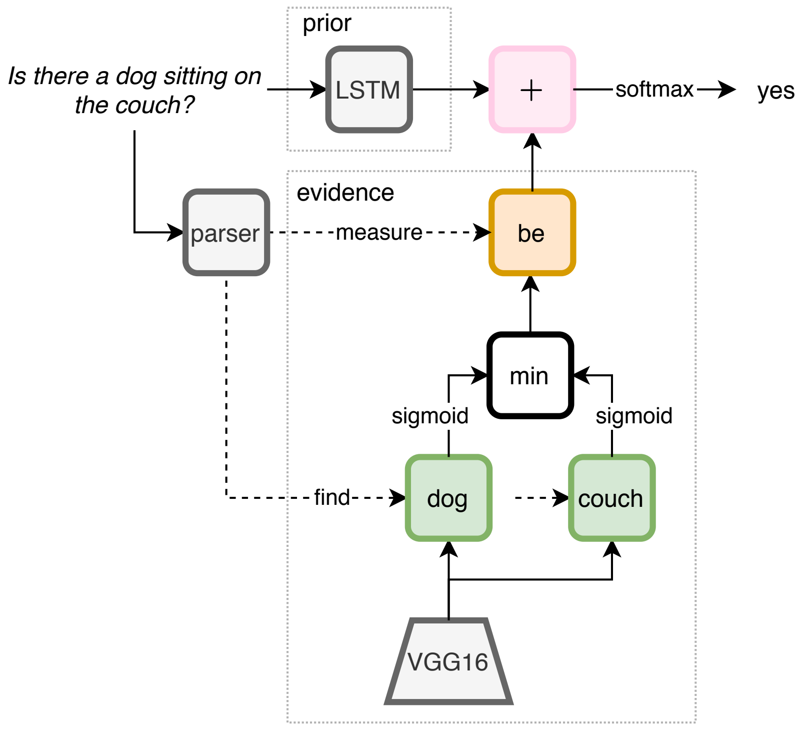


FIGURE 4.10: NMN architecture with neural data types. LSTM now provides a prior to the other branch and `Find` generates bounded soft masks, thus enabling the use of the minimum operator as the AND operator.

points otherwise. Additionally, we understand that the main functionality of the other modules is to provide such evidence or contradict the priors, and they should not have to care about other things. We therefore get rid of the geometric average and simply add the logits of both modules together, implementing this way an additive bias (see Figure 4.10) and also decreasing the gradient codependency between both branches. We cache these prior logits from the `QuestionEncoder` into an auxiliary data set.

Another interfacing point subject to redesign is the output of the `Find` module, which originally gives out *heatmaps or unnormalized attention*. Following the guidelines of section 4.2.1, we bound individual output values to the range

$(0, 1)$ by applying sigmoid activations and we re-define the output data type as soft masks. We can now rely on the operators *min* and *max* to implement AND and OR operators while avoiding extrapolation issues and favouring generalization. We have now a guarantee that values will always stay within the bounded range $(0, 1)$.

Furthermore, we leverage the low-memory footprint of modular training for exploring a wider range of architectural choices and hyperparameter configurations. For all modules we try batch sizes up to four times greater. For the `Find` module, we explore whether to use a bias term and we compare softmax and weighted average as possible candidates for the attention mechanism. Finally, we try different embedding sizes, number of hidden units and dropout rates for the modules `QuestionEncoder` and `Measure`. Such a rich exploration of the hyperparameter space would be near to infeasible in a monolithic scenario, due to the combinatorial explosion of configurations.

We show in Figure 4.11 the results of this hyperparameter search. You can observe now that the accuracy of the module `Find`'s SGM never goes below that of the `QuestionEncoder`. The combination of answers through the lens of priors has this effect, with the additional benefit of better focusing all module's functionality and avoiding the functional coupling that the geometric average introduced (this can be evidenced in `Measure` and `Describe`'s accuracies going up too). All in all, we achieve a 56.66% test accuracy, which improves 1.79 points over the end-to-end baseline (see Table 4.2).

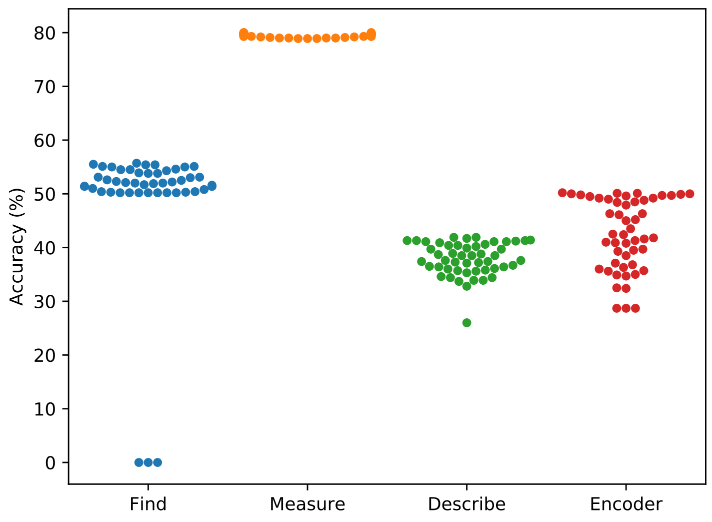


FIGURE 4.11: Point-wise representation of the modular hyperparameter search, including neural data types and extended hyperparameter configurations.

Model version	Accuracy
end-to-end baseline	54.87%
modular	54.49%
modular + data types	**56.66%**

TABLE 4.2: Test accuracies of different versions of the NMN architecture. Independent modular training does not hurt performance and implementing neural data types improves 1.79 points over the end-to-end baseline.

Element	Standard implementation	Neural data type
Attention map	heat map (ReLU)	mask (sigmoid)
Answer	logits	softmax values
AND	multiplication	minimum
OR	addition	maximum

TABLE 4.3: Pairwise comparison of modular interfaces. Standard implementations focus on improving end-to-end training convergence and type-based ones on compositional generalization (with neural data types).

4.3.3 Testing compositional generalization

In section 4.2.1 we hypothesize that neural data types help generalization by fostering compositional behaviour and in section 4.3.2 we show that it positively impacts performance on the VQA v1 dataset. However, this dataset does not explore compositionality in its full extent, since induced layouts rarely exceed a depth of two. For this reason we conduct here a series of experiments on the CLEVR dataset, which provides functional programs that can be mapped to modular layouts that are up to 22 modules in depth.

In order to test for compositional generalization, we train the modules jointly on layouts of a maximum depth of five. After the end-to-end training on shallow layouts, we test the modules on deeper layouts and see how much performance degrades with layout depth. Following this procedure we compare two different sets of modules which only differ on the interfaces, one of them having neural data types implemented and the other not. We specify these differences in Table 4.3.

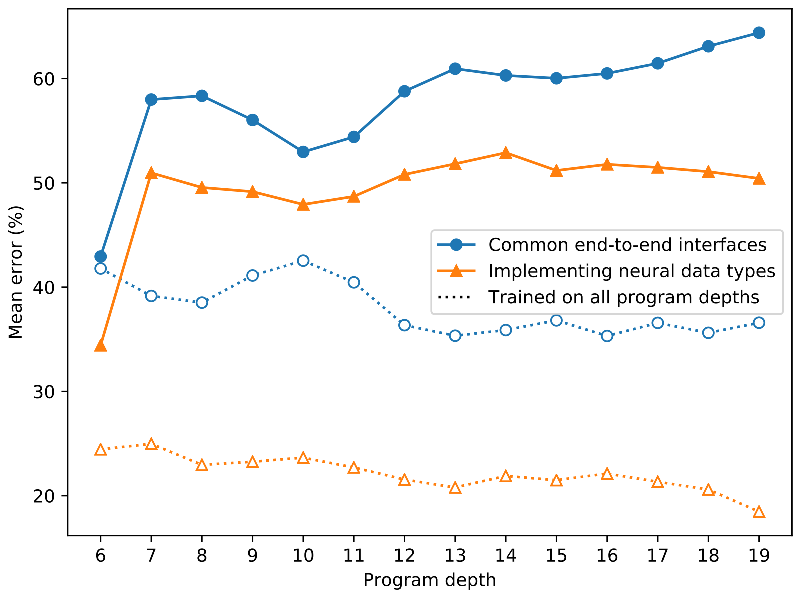
On the left in Figure 4.12 we show how neural data types help generalizing to layouts much deeper than those seen during training, although still using training images and sentences. On the upper plot we can additionally see the impact that the distributional shift of images and sentences has on performance. Shown as dotted lines, the error after training on all layouts also supports our hypothesis.

We contextualize these results by comparing our modular setup against a start-of-the-art monolithic architecture (FiLM, Perez et al., 2018). In Figure 4.13 we show the classification errors for both the FiLM network and our modular network with neural data types on the CLEVR validation set. Although FiLM is not modular and does not directly depend on program depth, the program depth is still a good heuristic for measuring the question complexity. FiLM performs much better on program depths seen during training, but its performance degrades faster and to a worse extent than its low-capacity modular competitor.

4.4 Main results

In this chapter, a methodology for independently training modules of a NN is proposed. This methodology relies on the assumption that the NN has been designed with modularity in mind in the first place. Moreover, our methodology strongly encourages the adoption of neural data types as a key element for improving compositional behaviour at the modular level.

In section 4.3.2 we have empirically shown that independent training of such neural modules is feasible, as long as inter-modular dependencies are identified (§4.1) and the corresponding measures are taken. We show that independent training results in modules with equivalent performance, with the additional benefit of enabling detailed modular inspection and analysis, and also making it possible to conduct training and hyperparameter optimization with a much lower



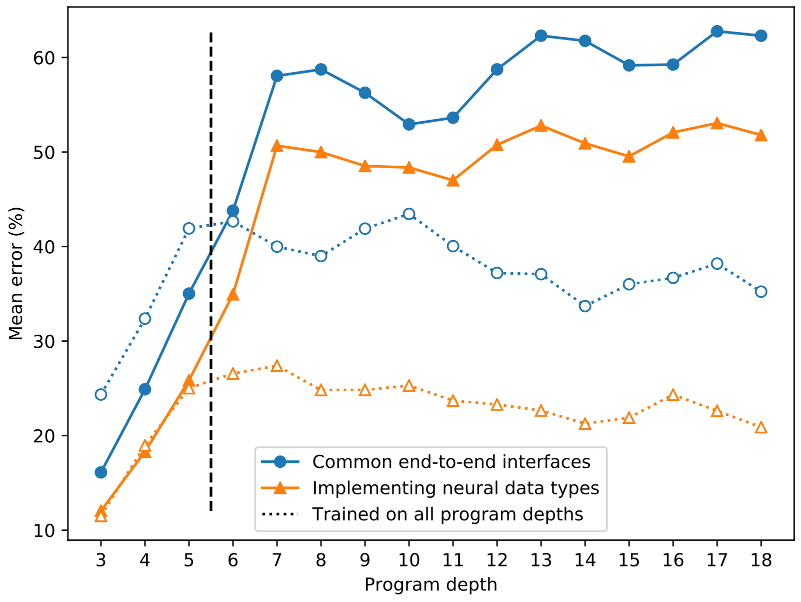


FIGURE 4.12: Mean classification error on training (top) and validation (bottom) data in function of the layout depth. A dashed vertical line separates depths seen from those unseen during training. For reference, we also show the error after training on all layout depths (dotted lines).

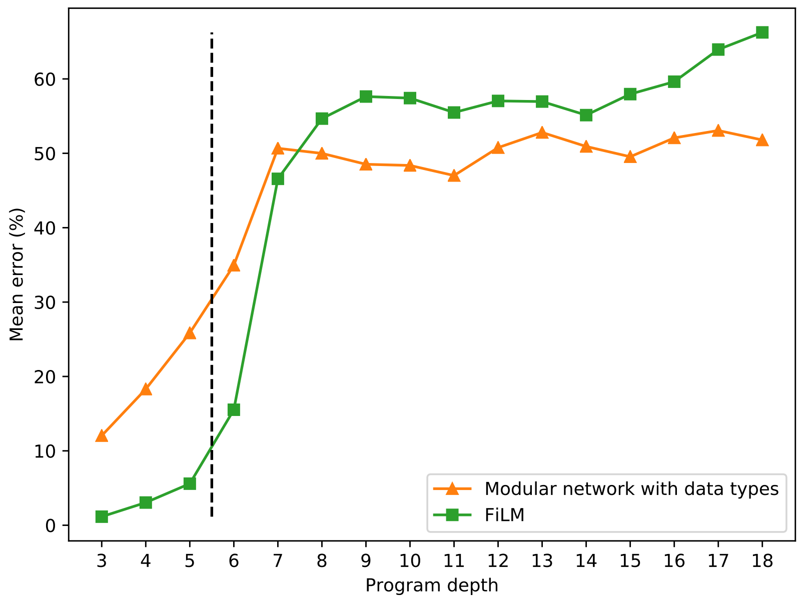


FIGURE 4.13: Generalization error in function of the program depth of our modular NN with neural data types and FiLM. Program depths after the dashed line were not seen during training.

computational budget and incurring in significantly lower computational costs.

One key contribution that enables independent modular training is the SGM (§4.2.3). We formalize *Learning by Role*, interpreting gradients as a directional resource for optimization and proposing the surrogate gradient hypothesis in consequence. We empirically demonstrate that one can imprint complex functionality into a neural module by representing all the functional constraints and requirements into a relatively simple auxiliary module, instead of relying on a complex layout of heavily parameterized post-processing modules, as it is traditionally required. We therefore present here the neural-networks equivalent of role play, by which kids learn through playing simple games the set of skills required in a more complex real-life scenario.

We have also shown that implementing neural data types significantly improves compositionality. Because NNs excel in the interpolation regime but tend to fail during extrapolation, it is in the interest of modular compositionality that data is represented within the boundaries of a narrowly enclosed space, which in turn facilitates the sufficient representation of the input manifold in the form of training samples.

Chapter 5

Conclusions, main contributions and future work

In this work we have revisited the concept of modularity in the context of DL and artificial NNs. By treating the learned functionality of a model as an additional non-explicit software layer, we have gained access to a novel perspective of the field, helping us better understand the scope of modularity in DL and realize that implementing modularity does not only require architectural design choices, but also training considerations. We have therefore brought the state of trained NNs closer to that of well-engineered software, which paves the way towards AI systems that are capable of solving a large diversity of problems in a structured way and maximizing knowledge reuse and maintainability.

Through the lens of neural-software modularity, and with the mentioned long-term goals in mind, we have revised the state of the art of NNs and DL in what relates to the general understanding of modularity, previous approaches to modular architectures and research results involving compositional behaviour. We have then proposed different methodological recipes for designing modular

NNs, as well as for designing and training neural modules which are capable of bringing high levels of compositionality through combinatorial extrapolation. We have shown that training the modules of a modular NN independently is possible and we have studied the benefits it brings, the challenges it poses and the possible ways to overcome them. In general, we have seen that modularity brings significant benefits in terms of scalability and maintainability, but also in other aspects.

5.1 Conclusions

Modularity is a concept that carries many implications with it, representing in fact a means to an end. Usually, a modular NN is expected to exhibit also compositional behaviour, which ideally requires the network to have capacity for re-combining knowledge or skills, and therefore exhibiting some degree of combinatorial generalization, which in turn boosts knowledge reuse. All these properties altogether bring a whole set of beneficial and desired features, like a positive impact in the learning efficiency and performance and generalization advantages. In general, the benefits aimed by modular NNs are very similar to those pursued by software engineering, and like it happens in computer science, achieving them is not a matter of following one single golden rule, but of carefully considering a set of guidelines and intuitions. Moreover, and following this analogy, there are many concepts originated from software engineering that can be applied in the field of DL. In this thesis we have mainly addressed coupling and cohesion, but there are probably many others to be leveraged.

Thinking about NNs in terms of learnable software definitely helps, unveiling a kind of software that cannot remain static as long as it receives learning signal, therefore originating new challenges and ways of working with neural modules. Within this context we can find software-engineering equivalents of established concepts. We can see weight sharing as a rough form of function definition, or the overfitting of one module to another module's output as a gradual expression of classical coupling. We can see how inputs and outputs can be somewhat standardized, bounded and their manifolds specified, in a way that resembles standard data types, and we can imagine how that might boost reuse. One particularly relevant revelation, product of this chain of thoughts, is the importance of functional composition. Most traditional interpretations of modularity focus on partitioning the input space and delegating those parts to several experts, but we can now see that such path can only provide as much computational complexity as parallel functions do, and the complexity of that aggregated function cannot be higher than that of the most complex expert. In contrast, by being able to feed outputs back to inputs and generate different functional compositions, we can have computational graphs of arbitrary complexity. In fact, we experimentally explored this idea to some extent in section 4.3.3.

In addition, we have also seen that achieving compositionality with NNs and end-to-end learning is quite challenging. Continual representation spaces and gradients tend to make NNs extremely sensitive to small variations, and the high precision of gradients contributes to this effect, amplifying subtle correlations up to the point of provoking overfitting. This thesis focus specially in this problem, trying to come up with ways of minimizing these effects of the training of neural modules while keeping their ability to learn effectively and perform well with good generalization. All in all, everything seems to indicate that continual representations and differentiable interfaces are not among the best tools when it comes to fostering combinatorial generalization, or even out-of-distribution generalization alone. We in fact show in section 3.2.2 a failed attempt to contain accumulated errors, which ended up increasing the training difficulty of modules and not improving their generalization. Despite these observations, the results presented in this thesis also indicate that there are ways of minimizing inter-modular coupling and improving compositionality whilst using continual and differentiable representations.

Independent modular training already helps reduce coupling by not allowing gradients to flow through the modular interface, therefore avoiding the trained module to overfit to the internal inner workings of the modules to which it connects. Additionally, the use of discrete and low-dimensional versions of training data increases the signal to noise ratio of the input and therefore reduces the chances of the module overfitting to spurious patterns in the data. In this regard, neural data types (§4.2.1) also compress the representational space of the data while keeping the relevant latent information untouched, which is to say that they increase the signal to noise ratio too. Furthermore, having neural data types be differentiable merely responds to the need of having gradient in order to train the module. If we were to have a way to train the corresponding module without direct gradients (e.g. through RL) we could rely on discrete data types and thus decrease the chances of overfitting even more [see Abolafia et al. (2020)]. Finally, one essential feature of neural data types is the introduction of domain boundaries, highlighting so an often forgotten aspect of the universal approximation theorem (Cybenko, 1989): in order to guarantee the compliance of the theorem, inputs must have *support in the unit hypercube* (i.e. inputs $x_i \in [0, 1]$). This condition is not strict, but it does in practice require the input values to be bounded to a range which can be efficiently covered during training.

On the other hand, in terms of computational costs, things are straightforward. We have clearly shown that the cost benefits of independent modular training are significant from the very first moment (Figures 3.6 and 4.9), in which training every module independently brings down costs already. The training can be several orders of magnitude faster than its monolithic counterpart in the best-case scenario (fully-supervised training), but even in a worst-case scenario the independent training of many modules is still advantageous because it can be carried out in parallel. Moreover, modular training enables a better usage of resources e.g. focusing the hyperparameter optimization in certain modules (Figure 4.11) or buffering more training samples in GPU. In the long run, we expect that the availability of pre-trained modules and other related modular-training resources makes the training costs fall even deeper, since a percentage of neural modules will be reused and therefore the training will only center on the novel ones. These results come at a time when end-to-end training of large monolithic models represents the main trend, with the consequent ever-growing training costs (Strubell, Ganesh, and McCallum, 2019; Thompson et al., 2021). The community has tried to bring these costs down, but mainly by reducing the need of labeled data and training supervision. The computational costs remain still there and, every time a new architectural element is introduced, new larger models are trained from scratch. In contrast, we have demonstrated that independent modular training is feasible and reliable, resulting in equivalent or better performance, and we have provided empirical examples in two different architectures and diverse configurations.

Among the most common challenges of independent modular training, we have identified those related to the availability of learning signal to be the most interesting (§4.1). In particular, the proposal of the SGM (§4.2.3) comes to answer one of those cases, in which the availability of gradient at the module's output depends on several other modules. By means of the SGM, based on the observed principle of *Learning by Role* (§4.2.2), a neural module can obtain an equivalent learning signal without having to run those expensive additional modules and just by having access to a relatively simple module. Additionally, the SGM serves as a relative indicator of the module's performance, helping the practitioner assess the stage of learning in which the module is at every moment. An explanation of the working principles of the SGM stands on two closely-related ideas. First, the *Learning by Role* acknowledges that the learning process of a module during end-to-end training is governed by many factors, being among the main ones the training data and the targets. In the absence of supervised labels, the learning signal is provided by the gradient coming from the modules using this module's outputs, and in absence of a specific data set, other modules provide for input too. In consequence, *Learning by Role* postulates that we can train a module as a byproduct of the scenarios it is exposed to during a multi-modular end-to-end training, which often happens by means of weight sharing. We find *Learning by Role* to be a powerful concept, which is widely applied in an implicit manner, yet rarely identified and exploited on purpose. Second, there is the observation that gradients are not an absolute learning signal, but a vector that indicates a relative change in output values or weights, and the identification of the possibility that a representation of the gradient function —the function that maps inputs and responses to gradients— might often be of little complexity. This latter aspect is informed by the observation of the relationship between games and their rules, road maps and directions, or even raw data and their compressed versions. In the case of games, chess for example, it is much simpler to represent the rules or constraints of the game than an algorithm which can always play the best move. In the same sense, we might only focus on remembering a couple of turning points in order to find always the way back home.

5.2 Main contributions

Because this work is based on a novel view of the field, the contributions do not only consist of methods and experiments, but also of this very view and the corresponding conclusions drawn from the current state of the art. From this perspective, by which DL models constitute an additional layer of software, the functionality is not encoded in a human-readable form and it can only be encoded through architectural design and a training procedure. Therefore, several new concepts and other theoretical contributions have been done too, as a result of this research.

A first contribution of this thesis is the review of the state of the art from the perspective of functional modularity and independent modular training (§2). Throughout this survey of the state of the art we reveal the many, diverse and often non-explicit ways in which modularity has been applied to NNs, and also that the motivations for seeking any form of modularity have been always present in the literature. Going by the motto "divide and conquer", most of the early work focused in various versions of model ensembling and modularization of learning algorithms, and only in the industry one could find very specialized modular networks trained under strong supervision. One important exception is the work of Waibel (1989) in merge-and-glue networks, which is to the best of our knowledge the oldest ancestor of our work.

Among the most recent literature, we found especially relevant the role of weight sharing as an explicit promoter of functional modularity, forcing NNs to reuse functionality in those places where it is applied. Weight sharing is also responsible for the first appearances of the *Learning by Role* phenomenon. We found distinct analyses of modularity and compositional behaviour in NNs (Hod et al., 2021; Béna and Goodman, 2021) that confirmed to us the loose relationship between structural and functional modularity. These studies, together with seminal works in relation to compositional behaviour and generalization (Cai, Shin, and Song, 2017; Li et al., 2020), pointed us to identify joint training and unbounded interfaces with input values presenting combinatorial explosion as the two main sources of coupling. Particularly, we emphasize the concurring work of Li et al. (2020), which agrees on our initial hypotheses about the importance of interface design, favouring discrete representations and constant input sizes, minimizing this way the degrees of sensitivity and highlighting the expressive power of functional composition.

In chapter 3 we do two main contributions: a general framework for modular intelligent systems and a study of the benefits and side-effects of independent modular training. The presented modular framework approaches tasks from the perspective of an agent interacting with an environment, in which one single task-specific module is in charge of composing the operators. Control module aside, all other modules and even the agent itself are abstracted via interfaces, which synchronize and standardize data between elements. This framework enables, in principle, to reuse operators across tasks and even encapsulate agents as operators of a higher-level agent, this way also opening the door to recursion. We exemplify this framework through its application to a list-sorting task and with this setup we conduct an analysis of the benefits and side-effects of independent modular training. Among the principal results, we show that independent modular training leads to learning times one order of magnitude shorter than with monolithic end-to-end training, despite requiring more training steps. We also show that modular training is much more stable, even when the task is artificially made harder, and that naive attempts to improve compositionality might actually result in worse generalization.

In chapter 4 we propose and validate a methodology for designing and training neural modules with high degrees of functional modularity, and therefore very akin to show compositional behaviour. We start by providing a taxonomy of modular dependencies that may affect independent modular training and inter-modular coupling (§4.1). In this section and along the chapter, we provide solutions to these dependencies and give examples of application.

A key element of the methodology we propose is the widespread use of neural data types, which we describe in section 4.2.1. With neural data types, we impose a series of features in modules' outputs that minimize the sources of coupling and thus foster functional and compositional generalization. The working principles of neural data types might be summarized into three main factors: faithfully represent the data type, keep inputs in the interpolation regime and reduce the amount of variation. All three factors contribute to avoiding issues with extrapolation and overfitting between modules.

Neural data types are in turn made possible by two additional contributions: the identification and characterization of the *Learning by Role* phenomenon (§4.2.2) and the proposal of the SGM (§4.2.3). Both concepts rely to some extent on the idea that the rules of a game are often very simple in comparison to the complexity of the game itself. That is how a kid can learn essential skills through role play and how we can make a module learn a particular functionality by building a minimalist representation of the task's constraints. We illustrate the proposal of the SGM with an example implementation, which is used for training a module of the NMN architecture (§4.3).

5.3 Future work

The theory and experimental results presented in this document open the door to diverse multiple paths for future research. Additionally, we have identified a series of caveats that may need further investigation. Finally, we would like to share our particular view on how modular intelligent systems could be developed.

Merely by recognizing the role of learned models as software components, many opportunities for contributing to software modularity emerge. A better understanding of how familiar concepts like coupling and cohesion can be applied at this level will improve functional modularity, knowledge reuse and compositional generalization.

The work on the modular framework that we propose in chapter 3 can be extended on many fronts, leveraging standardized interfaces in order to make intensive reuse of functionality. We find the possibilities that recursion offers especially interesting, as it is already a known fact that recursion is an elegant and powerful form of generalization (Cai, Shin, and Song, 2017). Also along this line, the field would greatly benefit from having a shared library of pre-trained modules. Solvers for specific sub-tasks could be encapsulated into new high-level operators. Existing operators could be improved without requiring control modules to be retrained and new controllers could be trained for solving new and more complex task at a higher abstraction level. We find this is a way in which knowledge reuse can be greatly improved and efforts can be more efficiently leveraged, building up complexity in a similar way than it is done with standard software components. It would also be interesting to see the application of this framework to more and more diverse tasks, hopefully contributing in the process to creating a public repository of pre-trained modules. In such repository, modules with standardized interfaces would be available for anyone to use, possibly including a differentiable version in addition. From this point on, it would be meaningful to measure how much this modular paradigm impacts the progress of the field in terms of speed and efficiency.

In section 3.2.3 we show that independent modular training requires a larger amount of training steps and nevertheless results in faster convergence than its monolithic end-to-end training counterpart. We also show that the gradient is apparently richer in the monolithic case and this seems to negatively affect the generalization properties of modules trained in isolation. It is relevant to study these matters further and see if these effects can be alleviated or compensated by any means.

The methodology presented in chapter 4 is a first attempt to design and train neural modules in a way that prevents inter-modular coupling and favours compositional generalization. However, there are still many open questions. For example, are there better ways of solving cases of dependent input, indirect supervision or codependent gradient? Are there other cases of inter-modular dependence so far unidentified? We found the cases of codependent gradient of especial difficulty, because they involve complex interactions between sources of data, and gradients are therefore entangled, too. All these matters require further investigation.

The current proposal of neural data types (§4.2.1) is heavily influenced by the need of backpropagating through the modules. However, even more strict data types and stronger generalization properties might be possible if modules are trained through reinforcement learning. Because modular training is much less expensive, RL methods might not involve a significant increase in overall cost and, on the other hand, they would allow the use of discrete or non-differentiable data types that better fit the targeted purpose.

Building SGMs is right now an artisan's work. One has to be very aware of what the target functionality is and find a way to represent it in terms of gradients and in a faithful way. There is therefore room for discovering methodologies or tools that guide this process, hopefully making it easier and more accessible to the research community. Additionally, SGMs work similarly (from the gradient point of view) to the discriminator in GAN settings. This implies that SGMs might also be learned and this could be investigated, probably through distillation of more complex ensembles into simpler modules. This could also be leveraged in the interest of performance and not just efficiency. Another line to investigate is the joint use of auxiliary losses and SGMs. A formal mathematical proof of the SGM and the principles that support it are also missing, and it seems that *Learning by Role* could be investigated further e.g. how much can complexity differ between the simulation and final tasks?

In our work, we used Monte-Carlo dropout for measuring uncertainty on the SGM. However, this is not an optimal method and the predictive variance of a diverse ensemble should work better according to the literature (Sekar et al., 2020). It is however not clear how this will affect the gradient and the SGM's building methodology. Another interesting thing to try might be to train differentiable modules that are equivalent to non-differentiable ones, just for the sake of using them in small end-to-end assemblies in replacement of commonly used non-differentiable algorithms or methods.

In section 4.3.2 we show how modularity can have a very positive impact in the hyperparameter search. It is left for future work to explore the design of a modular architecture from scratch for solving a multi-domain task and measure how the faster iteration at the modular level impacts final performance.

In section 4.3.3 we see the generalization advantage of a modular model against a state-of-the-art model (FiLM) in compositional generalization. Significant simplifications were made for instantiating the modular model, like directly using programs for layouts, and also the modules were extremely simple and low-capacity. Testing the performance of a modular network with neural data types plus controller is left for future work.

Finally, having a standard modular framework and being able to design and train modules capable of compositional generalization are definitely two key pieces of the puzzle, but the problem of finding good modular compositions or training controller policies is largely left for future research. We nevertheless envision a not-so-distant future in which repositories of pre-trained modules are publicly accessible. In this scenario, the amount of pre-trained modules available is so huge that the search for the right combination of modules can easily turn into a combinatorial optimization problem with no practical solution. This, however, can be greatly simplified through the leveraging of input and output data types, which set hard compatibility constraints for the search. After this initial prune is made, other finer-grained methods like attention can be used for estimating a probability distribution over the compatible modules. We are excited to think about this and other possibilities, and how future research on this topic will impact the field of DL and AI.

Appendix A

Resumen de la tesis en español

El concepto de modularidad es esencial en ingeniería, permitiendo no sólo dividir un problema complejo en partes más sencillas, sino que este proceso permite a su vez incrementar drásticamente la probabilidad de que una de estas piezas pueda ser reutilizada en el futuro. Esta reutilización del conocimiento es clave, pues permite el desarrollo incremental del mismo, en lugar de tener que empezar de cero cada vez. Sin embargo, el simple hecho de que un sistema esté construido mediante la composición de partes más sencillas (modularidad estructural) no implica necesariamente que las particiones del sistema favorezcan la composicionalidad funcional del mismo, ni mucho menos que estas partes puedan ser reutilizadas en nuevos sistemas.

En el campo del deep learning y las redes neuronales artificiales, la modularidad estructural es algo que se ha explotado ampliamente. Gracias a las librerías de cómputo en GPU y a las librerías que implementan las operaciones, capas y módulos más utilizados, la exploración de nuevos métodos y arquitecturas se ha establecido como un proceso altamente modular. Sin embargo, estos modelos de elevada modularidad arquitectónica son entrenados desde cero y de forma monolítica en la práctica totalidad de los casos. Si asemejáramos un modelo típico de deep learning a una computadora, podríamos decir que la parte hardware se construye mediante la recombinación de piezas existentes, pero el software es desarrollado en un solo fichero, específicamente para cada máquina y tarea. Esto es un problema no sólo desde el punto de vista práctico, sino tambien medioambiental.

Un obstáculo principal de cara a la modularización de la funcionalidad de los modelos de deep learning es su carácter aprendido y no programático. Es decir, que esta funcionalidad no se incorpora al modelo de forma altamente controlada, mediante programación manual, sino a través de un proceso indirecto de aprendizaje automático. Esto introduce diversos fenómenos que involucran el sobreajuste o sobreentrenamiento, haciendo que las partes del modelo se adapten a trabajar principalmente en la configuración de entrenamiento y que la funcionalidad se diluya o filtre por los distintos módulos. Además, la opacidad de las redes neuronales, característica de los modelos de tipo caja negra, presenta importantes desafíos de cara a la interpretación de su funcionalidad como software y a la aplicación de conceptos clásicos de ingeniería del software, como son el acoplamiento y la cohesión.

En esta tesis se presenta una visión del ámbito del deep learning desde una perspectiva de modularidad funcional, prestando especial atención al entrenamiento individualizado, reutilización y recombinación de módulos. Además, esta visión sirve de motivación principal en una serie de experimentos que tienen como objetivo el estudio de los diferentes aspectos que influyen en estos requisitos funcionales, así como un conjunto de propuestas metodológicas que facilitan el conseguimiento de dichos requisitos.

En primer lugar, presentamos una propuesta de marco arquitectónico para redes modulares y realizamos un estudio de factibilidad del entrenamiento individualizado y aislado de módulos. Los experimentos presentados en esta parte se centran en el análisis de los efectos adversos de este tipo de entrenamiento, así como el impacto del entrenamiento modular en el desempeño final y la generalización a nuevas instancias de distinto tamaño y complejidad. Así pues, esta sección de la tesis no sólo provee de un marco inicial de modularización de redes neuronales, sino que también acota los efectos y desafíos de base que presenta el entrenamiento individualizado de los módulos.

En una segunda parte de la tesis atajamos de forma específica el diseño y entrenamiento de módulos neuronales. Con el objetivo de optimizar el comportamiento composicional de los módulos, introducimos una serie de conceptos, metodologías y herramientas que ayudan a orientar el diseño de las interfaces modulares y a plantear el entrenamiento de los módulos, de forma que se puedan configurar con la mayor independencia posible. Las aportaciones se ejemplifican con la aplicación directa sobre una arquitectura de red neuronal modular de reciente impacto en el ámbito VQA (respuesta de preguntas basadas en imágenes).

A.1 Estudio del entrenamiento modular

En esta primera aproximación a la modularización de redes neuronales, presentamos una propuesta de marco estándar para la integración de módulos funcionales en un mismo modelo. En esta propuesta, los modelos son interpretados a modo de un agente que interactúa con un entorno, el cual le sirve no sólo de interfaz con la instancia del problema, sino también como instrumento para el almacenamiento de resultados temporales (Figura 3.1). El agente se divide en una serie de módulos, estando uno de ellos encargado del control y siendo el resto operadores, que pueden aplicarse a modo de acciones para cambiar el entorno. Tanto los operadores como el agente en sí cuentan con interfaces de entrada y salida, encargadas de abstraer la implementación del módulo y de adaptar el formato de los datos entre la representación externa e interna (Figura 3.2). Así pues, las distintas partes funcionales de los operadores se convierten en módulos altamente reutilizables, requiriendo únicamente la implementación de las respectivas interfaces. El módulo de control, por otra parte, sería el único módulo específico a la tarea en cuestión, ya que tendría como objetivo la selección de operadores a aplicar en función del estado actual del entorno.

Partiendo de este marco de arquitectura modular, realizamos un estudio de los requisitos básicos para el entrenamiento de los distintos módulos de forma independiente. De cara a realizar los distintos experimentos, planteamos un caso de estudio basado en la ordenación de listas de números enteros. Los resultados destacan la eficiencia del entrenamiento modular en términos de tiempo de cómputo. Aunque el entrenamiento monolítico es ligeramente más eficiente en cuanto al grado de mejora por iteración (Figura 3.7), el entrenamiento modular es casi un orden de magnitud más rápido (Figura 3.6), resultando menos costoso el entrenamiento con listas de mayor longitud si se realiza de esta manera. Como apunte adicional, detectamos que una técnica introducida en el entrenamiento modular, con la intención de minimizar la acumulación de errores, finalmente resulta en un incremento de la complejidad de la tarea y en un empeoramiento de la capacidad de generalización. Esta observación es reforzada por la inspección de los gradientes durante el entrenamiento, que muestra una mayor varianza de los mismos durante el entrenamiento monolítico (Figura 3.8).

A.2 Una metodología para el diseño de módulos neuronales altamente combinables

En este capítulo de la investigación nos centramos en los distintos factores y principios que influyen en el grado de composicionalidad de un módulo. Es decir, en qué grado se puede emplear el módulo de forma concatenada con otros módulos o él mismo, ensamblándose de formas nuevas y distintas a durante el entrenamiento. Comenzamos con una taxonomía de los distintos casos de dependencia que pueden presentarse en una red neuronal compuesta de módulos. La mayoría de estos casos viene caracterizada por aspectos logísticos, como el orden de entrenamiento, pero hay otros que presentan factores más complejos, como es el de la codependencia de gradiente. Para cada uno de estos casos damos propuestas de solución o indicaciones para mitigar su efecto.

Una de las aportaciones claves de esta metodología son las interfaces con tipos de datos neuronales. Estas interfaces surgen de reconocer una serie de fenómenos que juegan un papel clave en la generalización de las redes neuronales. Entre ellos, la elevada capacidad de interpolación, el mal desempeño en condiciones de extrapolación, el potencial del bias inductivo y el fenómeno de aprendizaje por rol. Este último, que describe la capacidad que tienen partes de una red neuronal para aprender funcionalidades necesarias para la tarea sin ser especificadas explícitamente por la función de pérdida global, motiva la propuesta del módulo de gradiente surrogado, el cual posibilita aproximar el gradiente de entrenamiento óptimo bajo condiciones de mínima complejidad computacional.

Mediante una serie de experimentos, demostramos empíricamente la validez del módulo de gradiente subrogado, tanto por su capacidad de servir de guía para el entrenamiento de otro módulo (Figura 4.5) como por su utilidad de cara a determinar el progreso del entrenamiento de forma relativa (Figuras 4.6 y 4.6). También mostramos las ventajas logísticas y computacionales del entrenamiento independiente de módulos, que permite entre otras cosas enfocar recursos a módulos concretos (Figura 4.11). En cuanto al aspecto composicional, no sólo demostramos la aplicación y beneficios sobre una arquitectura modular existente (Tabla 4.2), sino que también mostramos la superioridad del método de cara a la extrapolación a combinaciones de módulos nunca vistas y con una número de módulos muy superior al experimentado por el modelo durante el entrenamiento (Figura 4.13).

Bibliography

Abadi, Martín et al. (2016). "TensorFlow: a system for large-scale machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283.

Abolafia, Daniel A et al. (2020). "Towards modular algorithm induction". In: *arXiv preprint arXiv:2003.04227*.

Akyürek, Ekin, Afra Feyza Akyürek, and Jacob Andreas (2020). "Learning to recombine and resample data for compositional generalization". In: *arXiv preprint arXiv:2010.03706*.

Alet, Ferran, Tomás Lozano-Pérez, and Leslie P Kaelbling (2018). "Modular meta-learning". In: *Conference on Robot Learning*. PMLR, pp. 856–868.

Alom, Md Zahangir et al. (2019). "Recurrent residual U-Net for medical image segmentation". In: *Journal of Medical Imaging* 6.1, p. 014006.

Andreas, Jacob (2019). "Good-enough compositional data augmentation". In: *arXiv preprint arXiv:1904.09545*.

Andreas, Jacob et al. (2016). "Neural module networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 39–48.

Andrychowicz, Marcin et al. (2016). "Learning to learn by gradient descent by gradient descent". In: *Advances in neural information processing systems*, pp. 3981–3989.

Antol, Stanislaw et al. (2015). “Vqa: Visual question answering”. In: *Proceedings of the IEEE international conference on computer vision*, pp. 2425–2433.

Artetxe, Mikel and Holger Schwenk (2019). “Massively multilingual sentence embeddings for zero-shot cross-lingual transfer and beyond”. In: *Transactions of the Association for Computational Linguistics* 7, pp. 597–610.

Auda, Gasser and Mohamed Kamel (1999). “Modular neural networks: a survey”. In: *International journal of neural systems* 9.02, pp. 129–151.

Battaglia, Peter W et al. (2018). “Relational inductive biases, deep learning, and graph networks”. In: *arXiv preprint arXiv:1806.01261*.

Béna, Gabriel and Dan FM Goodman (2021). “Extreme sparsity gives rise to functional specialization”. In: *arXiv preprint arXiv:2106.02626*.

Bernstein, Jeremy et al. (2018). “signSGD: Compressed optimisation for non-convex problems”. In: *International Conference on Machine Learning*. PMLR, pp. 560–569.

Bommasani, Rishi et al. (2021). “On the opportunities and risks of foundation models”. In: *arXiv preprint arXiv:2108.07258*.

Brock, Andrew, Jeff Donahue, and Karen Simonyan (2019). “Large Scale GAN Training for High Fidelity Natural Image Synthesis”. In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=B1xsqj09Fm.

Brown, Tom et al. (2020). “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33, pp. 1877–1901.

Bunel, Rudy et al. (2018). “Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis”. In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=H1Xw62kRZ.

Burridge, Robert R, Alfred A Rizzi, and Daniel E Koditschek (1999). “Sequential composition of dynamically dexterous robot behaviors”. In: *The International Journal of Robotics Research* 18.6, pp. 534–555.

Cai, Jonathon, Richard Shin, and Dawn Song (2017). “Making neural programming architectures generalize via recursion”. In: *arXiv preprint arXiv:1704.06611*.

Caruana, Rich (1997). “Multitask learning”. In: *Machine learning* 28.1, pp. 41–75.

Castillo-Bolado, David, Cayetano Guerra-Artal, and Mario Hernández-Tejera (2019). “Modularity as a means for complexity management in neural networks learning”. In: *arXiv preprint arXiv:1902.09240*.

— (2021). “Design and independent training of composable and reusable neural modules”. In: *Neural Networks* 139, pp. 294–304.

Chen, Ke (2015). “Deep and modular neural networks”. In: *Springer Handbook of Computational Intelligence*. Springer, pp. 473–494.

Chen, Ting et al. (2020). “A simple framework for contrastive learning of visual representations”. In: *International conference on machine learning*. PMLR, pp. 1597–1607.

Cho, Kyunghyun et al. (2014). “On the properties of neural machine translation: Encoder-decoder approaches”. In: *arXiv preprint arXiv:1409.1259*.

Cybenko, George (1989). “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of control, signals and systems* 2.4, pp. 303–314.

D. Erman, Lee et al. (June 1980). “The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty”. In: *ACM Comput. Surv.* 12, pp. 213–253. DOI: 10.1145/356810.356816.

Dai, Hanjun (2020). “Learning neural algorithms with graph structures”. PhD thesis. Georgia Institute of Technology.

Dai, Hanjun et al. (2018). “Learning steady-states of iterative algorithms over graphs”. In: *International conference on machine learning*. PMLR, pp. 1106–1114.

Devlin, Jacob et al. (2018). “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805*.

Flennerhag, Sebastian et al. (2021). “Bootstrapped meta-learning”. In: *arXiv preprint arXiv:2109.04504*.

Gidaris, Spyros, Praveer Singh, and Nikos Komodakis (2018). “Unsupervised representation learning by predicting image rotations”. In: *arXiv preprint arXiv:1803.07728*.

Gleave, Adam et al. (2020). “Adversarial Policies: Attacking Deep Reinforcement Learning”. In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=HJgEMpVFwB.

Goodfellow, Ian et al. (2014). “Generative Adversarial Nets”. In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., pp. 2672–2680. URL: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

Goyal, Anirudh et al. (2019). “Recurrent independent mechanisms”. In: *arXiv preprint arXiv:1909.10893*.

Graves, Alex et al. (2016). “Hybrid computing using a neural network with dynamic external memory”. In: *Nature* 538.7626, pp. 471–476.

Grefenstette, Edward et al. (2015). “Learning to transduce with unbounded memory”. In: *Advances in neural information processing systems* 28, pp. 1828–1836.

Gupta, Nitish et al. (2019). “Neural module networks for reasoning over text”. In: *arXiv preprint arXiv:1912.04971*.

Hanin, Boris and Mark Sellke (2017). “Approximating continuous functions by relu nets of minimal width”. In: *arXiv preprint arXiv:1710.11278*.

He, Kaiming et al. (2016). “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Hebb, Donald Olding (1949). “The organization of behavior; a neuropsycholocigal theory.” In: *A Wiley Book in Clinical Psychology.*, pp. 62–78.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long short-term memory”. In: *Neural computation* 9.8, pp. 1735–1780.

Hod, Shlomi et al. (2021). “Detecting Modularity in Deep Neural Networks”. In: *arXiv preprint arXiv:2110.08058*.

Hong, Khay Boon (2022). “U-Net with ResNet Backbone for Garment Landmarking Purpose”. In: *arXiv preprint arXiv:2204.12084*.

Hrycej, Tomas (1992). *Modular learning in neural networks: a modularized approach to neural network classification.* John Wiley & Sons, Inc.

Hu, Ronghang et al. (2017). “Learning to reason: End-to-end module networks for visual question answering”. In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 804–813.

Hu, Ronghang et al. (2018). “Explainable neural computation via stack neural module networks”. In: *Proceedings of the European conference on computer vision (ECCV)*, pp. 53–69.

Hupkes, Dieuwke et al. (2019). “The compositionality of neural networks: integrating symbolism and connectionism”. In: *arXiv preprint arXiv:1908.08351*.

Jacobs, Robert A, Michael I Jordan, and Andrew G Barto (1991). “Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks”. In: *Cognitive science* 15.2, pp. 219–250.

Jacobs, Robert A et al. (1991). “Adaptive mixtures of local experts”. In: *Neural computation* 3.1, pp. 79–87.

Jaegle, Andrew et al. (2021). “Perceiver: General perception with iterative attention”. In: *arXiv preprint arXiv:2103.03206*.

Johnson, Justin et al. (2017). “Clevr: A diagnostic dataset for compositional language and elementary visual reasoning”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2901–2910.

Joulin, Armand and Tomas Mikolov (2015). “Inferring algorithmic patterns with stack-augmented recurrent nets”. In: *Advances in neural information processing systems* 28, pp. 190–198.

Kah, Ang Heng et al. (1995). “Smart air-conditioning system using multilayer perceptron neural network with a modular approach”. In: *Proceedings of ICNN’95-International Conference on Neural Networks*. Vol. 5. IEEE, pp. 2314–2319.

Kingma, D. P. and J. Ba (Dec. 2014). “Adam: A Method for Stochastic Optimization”. In: *ArXiv e-prints*. arXiv: 1412.6980.

Kirsch, Louis, Julius Kunze, and David Barber (2018). “Modular networks: Learning to decompose neural computation”. In: *arXiv preprint arXiv:1811.05249*.

Klein, Dan and Christopher D. Manning (2003). “Accurate Unlexicalized Parsing”. In: *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*. ACL ’03. USA: Association for Computational Linguistics, 423–430. DOI: 10.3115/1075096.1075150. URL: https://doi.org/10.3115/1075096.1075150.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25, pp. 1097–1105.

Kurach, Karol, Marcin Andrychowicz, and Ilya Sutskever (2015). "Neural random-access machines". In: *arXiv preprint arXiv:1511.06392*.

Ladousse, G.P. and A. Maley (1987). *Role Play*. Oxford English. OUP Oxford. ISBN: 9780194370950.

LeCun, Yann et al. (1989). "Handwritten digit recognition with a back-propagation network". In: *Advances in neural information processing systems* 2.

Li, Yujia et al. (2020). "Strong generalization and efficiency in neural programs". In: *arXiv preprint arXiv:2007.03629*.

Liška, Adam, Germán Kruszewski, and Marco Baroni (2018). "Memorize or generalize? searching for a compositional rnn in a haystack". In: *arXiv preprint arXiv:1802.06467*.

Loula, Joao, Marco Baroni, and Brenden M Lake (2018). "Rearranging the familiar: Testing compositional generalization in recurrent networks". In: *arXiv preprint arXiv:1807.07545*.

Lu, Bao-Liang and Masami Ito (1999). "Task decomposition and module combination based on class relations: a modular neural network for pattern classification". In: *IEEE Transactions on Neural networks* 10.5, pp. 1244–1256.

Lu, Zhou et al. (2017). "The Expressive Power of Neural Networks: A View from the Width". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., pp. 6231–6239. URL: `http://papers.nips.cc/paper/7203-the-expressive-power-of-neural-networks-a-view-from-the-width.pdf`.

Lutter, Michael et al. (2021). "Differentiable physics models for real-world offline model-based reinforcement learning". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 4163–4170.

Mao, Jiayuan et al. (2019). “The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision”. In: *arXiv preprint arXiv:1904.12584*.

Mecklenburg, Klaus et al. (1992). “Neural control of autonomous vehicles”. In: *[1992 Proceedings] Vehicular Technology Society 42nd VTS Conference-Frontiers of Technology*. IEEE, pp. 303–306.

Montague, Richard (1970). “Universal grammar”. In: *Theoria* 36.3, pp. 373–398.

Mordido, Goncalo, Matthijs Van Keirsbilck, and Alexander Keller (June 2020). “Monte Carlo Gradient Quantization”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

Novak, Roman et al. (2018). “Sensitivity and Generalization in Neural Networks: an Empirical Study”. In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=HJC2SzZCW.

Ontañón, Santiago et al. (2021). “Making Transformers Solve Compositional Tasks”. In: *arXiv preprint arXiv:2108.04378*.

Paszke, Adam et al. (2019). “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32, pp. 8026–8037.

Perez, Ethan et al. (2018). “FiLM: Visual Reasoning with a General Conditioning Layer”. In: *AAAI*.

Radford, Alec et al. (2018). *Improving language understanding by generative pre-training (2018).*

Ramesh, Aditya et al. (2021). “Zero-shot text-to-image generation”. In: *arXiv preprint arXiv:2102.12092*.

Ramesh, Aditya et al. (2022). "Hierarchical text-conditional image generation with clip latents". In: *arXiv preprint arXiv:2204.06125*.

Reed, Scott and Nando De Freitas (2015). "Neural programmer-interpreters". In: *arXiv preprint arXiv:1511.06279*.

Reed, Scott et al. (2022). "A Generalist Agent". In: *arXiv preprint arXiv:2205.06175*.

Ritter, Helge and Teuvo Kohonen (1989). "Self-organizing semantic maps". In: *Biological cybernetics* 61.4, pp. 241–254.

Rosa, Marek et al. (2019). "BADGER: Learning to (Learn [Learning Algorithms] through Multi-Agent Communication)". In: *arXiv preprint arXiv:1912.01513*.

Rosenbaum, Clemens et al. (2019). "Routing networks and the challenges of modular and compositional computation". In: *arXiv preprint arXiv:1904.12774*.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1985). *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science.

Sanchez-Gonzalez, Alvaro et al. (2018). "Graph networks as learnable physics engines for inference and control". In: *International Conference on Machine Learning*. PMLR, pp. 4470–4479.

Santoro, Adam et al. (2017). "A simple neural network module for relational reasoning". In: *arXiv preprint arXiv:1706.01427*.

Schrittwieser, Julian et al. (2020). "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839, pp. 604–609. ISSN: 0028-0836. DOI: 10.1038/s41586-020-03051-4.

Sekar, Ramanan et al. (2020). "Planning to explore via self-supervised world models". In: *International Conference on Machine Learning*. PMLR, pp. 8583–8592.

Shlomi, Jonathan, Peter Battaglia, and Jean-Roch Vlimant (2020). “Graph neural networks in particle physics”. In: *Machine Learning: Science and Technology* 2.2, p. 021001.

Shorten, Connor and Taghi M Khoshgoftaar (2019). “A survey on image data augmentation for deep learning”. In: *Journal of big data* 6.1, pp. 1–48.

Silver, David et al. (2016). “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587, pp. 484–489.

Silver, David et al. (2018). “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419, pp. 1140–1144. DOI: 10.1126/science.aar6404.

Simonyan, Karen and Andrew Zisserman (Sept. 2014). “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *arXiv e-prints*, arXiv:1409.1556, arXiv:1409.1556. arXiv: 1409.1556 [cs.CV].

Sohl-Dickstein, Jascha (2021). *Learned optimizers: why they're the future, why they're hard, and what they can do now.* URL: https://www.iarai.ac.at/events/learned-optimizers-why-theyre-the-future-why-theyre-hard-and-what-they-can-do-now/ (visited on 08/10/2022).

Strubell, Emma, Ananya Ganesh, and Andrew McCallum (2019). “Energy and policy considerations for deep learning in NLP”. In: *arXiv preprint arXiv:1906.02243*.

Szabó, Zoltan (2012). “The case for compositionality”. In: *The Oxford handbook of compositionality* 64, p. 80.

Szegedy, Christian et al. (2015). “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9.

Tanner, Martin A and Wing Hung Wong (1987). “The calculation of posterior distributions by data augmentation”. In: *Journal of the American statistical Association* 82.398, pp. 528–540.

Theodoridis, Sergios and Konstantinos Koutroumbas (2006). *Pattern recognition.* Elsevier.

Thompson, Neil C et al. (2021). “Deep learning’s diminishing returns: the cost of improvement is becoming unsustainable”. In: *IEEE Spectrum* 58.10, pp. 50–55.

Thoppilan, Romal et al. (2022). “Lamda: Language models for dialog applications”. In: *arXiv preprint arXiv:2201.08239.*

Vaswani, Ashish et al. (2017). “Attention is all you need”. In: *Advances in neural information processing systems*, pp. 5998–6008.

Waibel, Alex (1989). “Modular construction of time-delay neural networks for speech recognition”. In: *Neural computation* 1.1, pp. 39–46.

Wang, Lin-Cheng, Nasser M Nasrabadi, and Sandor Der (1997). “Asymptotical analysis of a modular neural network”. In: *Proceedings of International Conference on Neural Networks (ICNN’97).* Vol. 2. IEEE, pp. 1019–1022.

Yoon, Jinsung et al. (2020). “Vime: Extending the success of self-and semi-supervised learning to tabular domain”. In: *Advances in Neural Information Processing Systems* 33.

Yosinski, Jason et al. (2015). “Understanding neural networks through deep visualization”. In: *arXiv preprint arXiv:1506.06579.*

Zhang, Si et al. (2019). “Graph convolutional networks: a comprehensive review”. In: *Computational Social Networks* 6.1, pp. 1–23.

Zoph, Barret et al. (2018). “Learning transferable architectures for scalable image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710.