Designing a Performant Ablation Study Framework for PyTorch

Alessio Molinari

Author

Alessio Molinari <amolina@kth.se>
EIT Digital, Data Science Track
KTH Royal Institute of Technology

Host Company

Logical Clocks AB
Stockholm, Sweden

Examiner

Prof. Amir H. Payberah
KTH Royal Institute of Technology

Supervisors

Sina Sheikholeslami (Academic Supervisor)

Moritz Meister (Industrial Supervisor)

KTH Royal Institute of Technology

Abstract

PyTorch is becoming a really important library for any deep learning practitioner, as it provides many low-level functionalities that allow a fine-grained control of neural networks from training to inference, and for this reason it is also heavily used in deep learning research, where ablation studies are often conducted to validate neural architectures that researchers come up with. To the best of our knowledge, Maggy is the first open-source framework for asynchronous parallel ablation studies and hyperparameter optimization for TensorFlow, and in this work we added important functionalities such as the possibility to execute ablation studies on PyTorch models as well as the generalization of feature ablation on any data type. This work also shows the main challenges and interesting points of developing a framework on top of PyTorch and how these challenges have been addressed in the extension of Maggy.

Keywords

Ablation Study, PyTorch, Neural Architecture Search, Spark

Abstract

PyTorch blir ett oerhört viktigt bibliotek för alla utövare inom djupinlärning, detta eftersom PyTorch innehåller flertalet lågnivåfunktioner som möjliggör en finkorning kontroll av neurala nätverk - från träning till inferens. Av den anledningen används PyTorch också kraftigt i forskning om djupinlärning, där ablationsstudier ofta genomförs för att validera neurala arkitekturer som forskare framtagit. Så vitt vi vet är Maggy det första open-source ramverk för asynkrona parallella ablationsstudier och hyperparameteroptimering för TensorFlow. I detta arbete har vi lagt till viktiga funktioner såsom möjligheten att utföra ablationsstudier på PyTorch-modeller samt generalisering av funktionsablation för alla datatyper. Detta arbete upplyser också dem viktigaste utmaningarna och mest intressanta punkterna för att utveckla en ram ovanpå PyTorch och hur dessa utmaningar har hanterats i förlängningen av Maggy.

Nyckelord

Ablation Study, PyTorch, Neural Architecture Search, Spark

Acknowledgements

To my friends in Stockholm, Nice and all around the world that have accompanied me throughout this life-changing two-year journey.

To my family, for always being there.

A special thank you goes to my examiner Amir and my supervisors Sina, Moritz and Jim for teaching me a lot and giving me useful feedback.

Acronyms

AI	Artitificial Intelligence
ML	Machine Learning
NAS	Neural Architecture Search
AutoML	Automated Machine Learning
ANN	Artificial Neural Network
ICT	Information and Communication Technology
CNN	Convolutional Neural Network
LSTM	Long-Short Term Memory
NLP	Natural Language Processing
HO	Hyperparameter Optimization
HDFS	Hadoop File System
GPU	Graphics Processing Unit
LOCO	Leave One Component Out
RNN	Recurrent Neural Network
DAG	Directed Acyclic Graph
API	Application Programming Interface
IT	Information Technology
XOR	Exclusive OR
CUDA	Compute Unified Device Architecture
SQL	Structured Query Language
RPC	Remote Procedure Call
VCS	Version Control System

RDD Resilient Distributed Dataset

CPU Central Processing Unit

FS File System

DRY Don’t Repeat Yourself

Contents

List of Figures

Listings

Chapter 1

Introduction

In this chapter we are going to introduce the key concepts of this thesis: we start by introducing problem, motivation, purpose and goal for this project. We will also present sections on ethics and sustainability, methodology, as well as the delimitations and outline for this work.

1.1 Problem

In *Machine Learning (ML)*, *ablation studies* are the removal of one or more components from a neural architecture or its training dataset. Their main purpose is to show how a certain component of the system impacts some relevant metrics (e.g. the accuracy of the model). Ablation studies are of primary importance in ML research as well as in real-world applications since they are useful to validate and justify proposed neural network models and datasets. However, they are not carried out very frequently because they require a lot of code changes, and in general they are considered time-consuming since they require retraining a model multiple times, especially if many configurations need to be validated. In general, for every new configuration that one wishes to test, it is necessary to make some code modifications and then retrain the model from scratch. On top of this, researchers need to keep track of all the conducted experiments with details on what has been changed in the neural model and the various metrics that have been collected. This can result in a lot of work that can be automated with an adequate framework. In this thesis work we will introduce Maggy [23]: a framework that aims to remove most of that complexity, automating the model training process and leaving to the user the only task of specifying the ablation process that needs to be executed.

Maggy was developed in 2019 by Logical Clocks AB [21] and provides *Application Programming Interfaces (APIs)* for ablation studies and hyperparameter search in TensorFlow [1]. We believe that providing data scientists with a tool to test their models in a flexible and easy way will produce higher-quality research and, furthermore, it will save them a considerable amount of time that otherwise should be dedicated to manual model search.

1.2 Motivation

Many influential papers [11, 19, 38, 43] contain sections on ablation studies, showing that they are indeed a useful tool to gain some more information about complex models. Some of them describe simple removals or neural network layers (or groups of layers), while other papers show complex ablations that involves loss functions or small details in the neural architecture. Some of these examples on how broad ablation studies can be are reported in section 5.3.

Even though TensorFlow is still a popular framework for deep learning, PyTorch has also gained a lot of attention in the last recent years (especially in academia, as we can see in Figure 1.2.1), because of some interesting features that allow low-level control of neural models. One of the main reasons of PyTorch's success is certainly the fact the it is a *Pythonic* framework [35]: this means that it exposes a familiar interface for programmers that are already familiar with Python (e.g. the model training loop is a normal Python program and the necessary data structures are simple Python classes that the user needs to subclass). On the other hand, in TensorFlow, the objects and functions exposed to the user hide most of the complexity underneath the training process, which makes it very easy and intuitive to use, but at the same time it does not provide the same fine-grained control that PyTorch has (we will discuss this concept more in detail in Section 3.3.1). Regarding the optimization, even though it is difficult to quantify with precision which framework is faster, PyTorch is appreciated for the fact its core functionalities are written in C++ to ensure high performance. Specifically, the control flow is handled by a C++ core and the data flow (tensor management) is optimized through *Compute Unified Device Architecture (CUDA)* [32], which executes tensor operations on one or more *Graphics Processing Units (GPUs)* when these are available (this mechanism is transparent to the user). To the best of our knowledge,

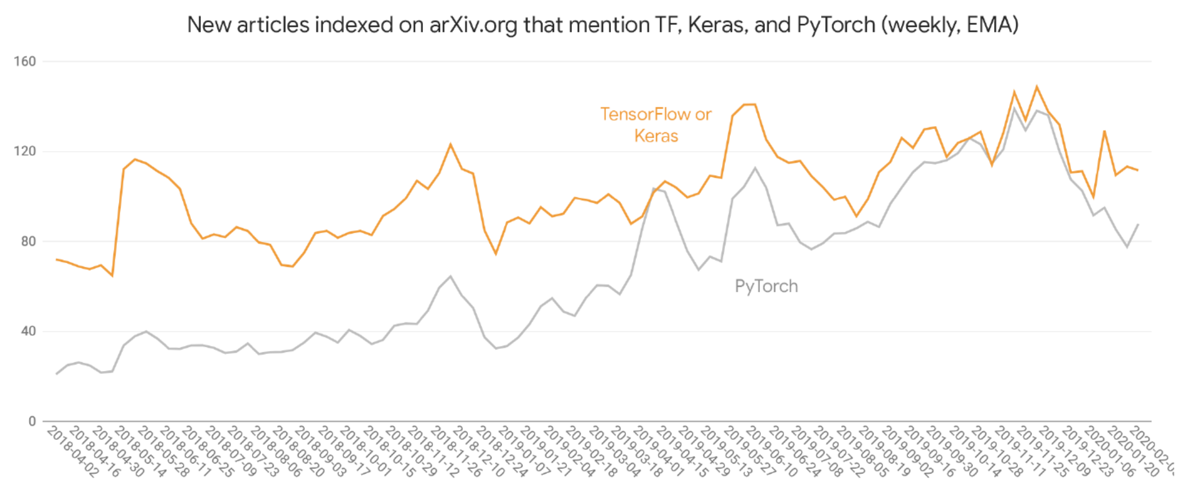

Figure 1.2.1: TensorFlow and PyTorch mentions in arXiv.org over time [37]

there was no framework for ablation studies in PyTorch [35] prior to this work, and that is why my supervisors and I decided to explore the possibility of such techniques, eventually integrating this functionality into Maggy [23].

1.3 Goal and Research Question

The goal of this thesis work is to produce a framework for ablation studies that is simple to use, requires minimal code changes compared to ordinary PyTorch code, is performant and can be used in a distributed Spark [34] environment. Furthermore, since an API for Maggy is already available, we also kept that interface unchanged to allow Maggy users to switch from PyTorch to TensorFlow in a seamless fashion. This work also aims to explore the strength as well as the limitations of PyTorch, illustrating the main challenges and interesting points in the development of Maggy: by analyzing PyTorch in depth we shed some light on the difficulties that stem from building a framework on top of it, as well as some features that can be harnessed to make it flexible and easy to use. Sections 3.2 and 3.3.1 will reprise and elaborate these points to show exactly what has been done in the design and development of this framework. The research question can be formulated as follows: “Can we provide an intuitive API for PyTorch users to efficiently conduct ablation studies with minimal code changes compared to normal PyTorch code?”

1.4 Ethics and Sustainability

Even though sustainability is a topic that is not often mentioned together with *Artitificial Intelligence (AI)*, it has major importance nowadays more than ever. Schwartz et al. published in 2019 a paper on "Green AI" [38] which points out that throughout the development of AI, the size of neural networks have increased at a pace even faster than Moore's law. Nowadays, to produce accurate AI models we often need a huge amount of GPUs and energy. Ironically, these models should be inspired to the neural mechanisms in our brain which, on the contrary, are extremely efficient. Their work suggest that research on AI so far has not been organic and methodical, but rather rushed and only focused to publish many new models as fast as possible, as it is confirmed also by Lindauer & Hutter [19]. What deep learning research needs in this moment is first of all more common guidelines on how to build and evaluate new models, and secondly it needs a way to make models more efficient and accessible.

We really do not perceive the link between energy inefficiency and *Information Technology (IT)*, maybe because we use terms like "cloud", "intelligence" and others that sound very positive. However, in September 2018, Nature reported that the energy needed to run and cool down only data centers added up to 1% of the total world energetic consumption. If we consider also personal devices (like computers, tablets and mobile phones) this share increases up to 2% [12]. Recently, also Cryptocurrencies mining has taken a toll on this numbers, and only in these years we begin to see statistics on the specific consumption of AI models because we just realized the amount of energy that they demand. In Figure 1.4.1 we can see a prediction of the energy demand due to *Information and Communication Technology (ICT)* according to Nature.

Automating *Neural Architecture Search (NAS)* and *Hyperparameter Optimization (HO)* is a first crucial step to build efficient models because humans are just not good at performing these tasks [37]. Secondly, we have to make this processes efficient: an exhaustive search that runs for months trying all possible alternatives for a certain neural architecture is fully automated but completely inefficient. We have to harness domain knowledge for specific problems to optimize the resources that we have. Early stopping is a possible way to achieve this, by stopping experiments that most likely won't give rise to accurate models, therefore saving training time. Support for early

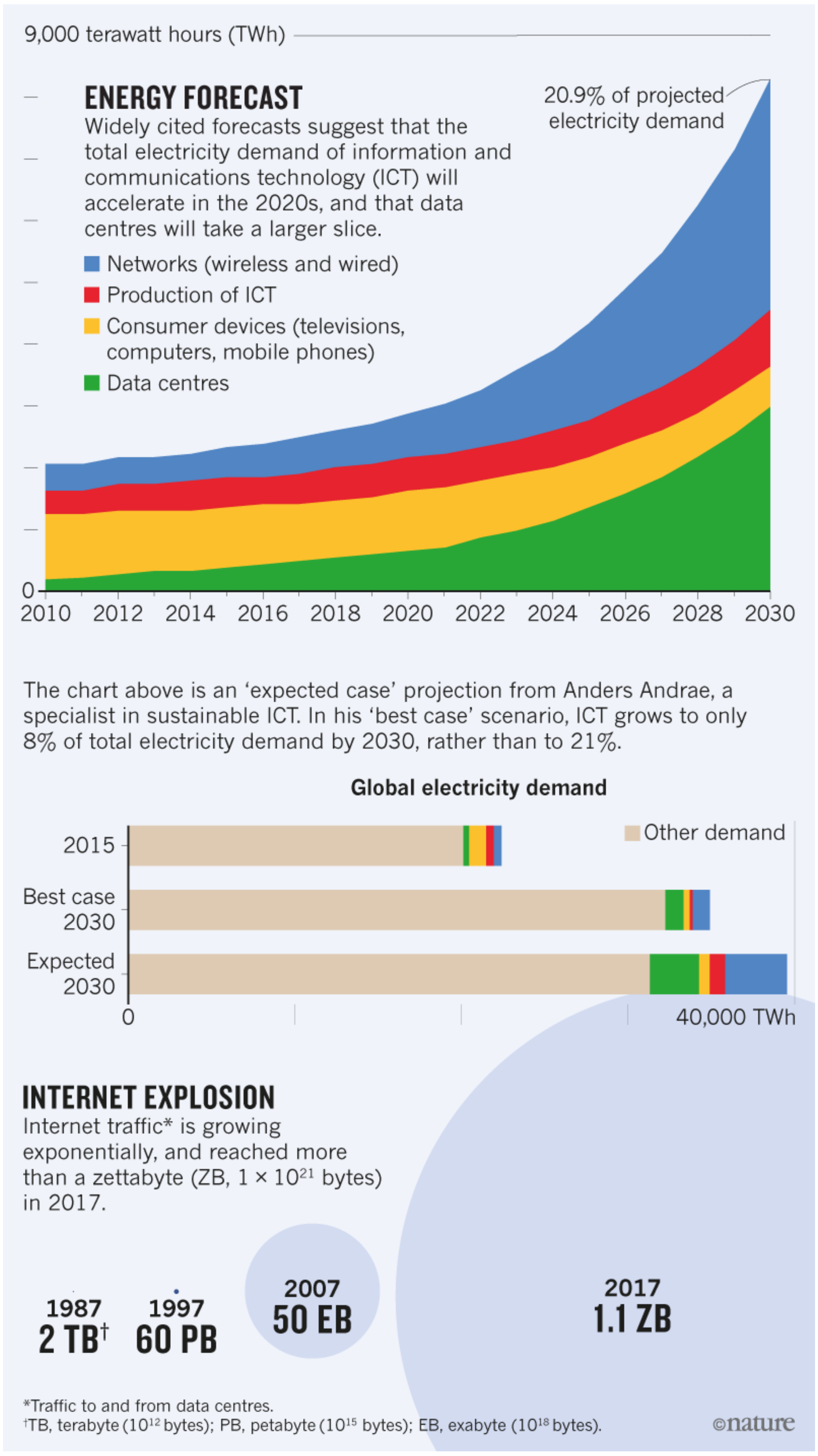

Figure 1.4.1: Prediction of energy demand in ICT [31]

stopping is already implemented in Maggy for HO experiments [22].

1.5 Methodology

A common thesis work involves devising an experiment, collecting data and then validating the results obtained. This thesis presents the design and implementation of a framework that is conceived to be used by other developers and therefore does not follow the classic experimental methodology (collecting information, devising an experiment and showing the results). Here are the main steps that defined the realization of this degree project:

- Recognize the need for ablation study frameworks as well as the value of PyTorch in deep learning research and identify a gap in academic literature (the absence of ablation study frameworks for PyTorch).
- Study PyTorch extensively, especially the most common ways of using it, since one of the goals of this project is to build a tool that would be easily usable by developers that already know how to write PyTorch code. This aspect will be elaborated more in Chapter 3.
- Get information on the state of the art in ablation studies and NAS.
- Define the design principles: ease of use, efficiency and minimal code changes compared to PyTorch code (described more in detail in Section 3.1).
- Identify the main problems in feature and model ablation in PyTorch.
- Extend Maggy with model and feature ablation for PyTorch following its original programming model [29]. In this case, an alternative API was also studied and presented in the thesis in Section 3.5.
- Evaluating the framework with different neural architectures and ablation studies.

In the scope of this thesis we will look at ablation studies that involve mainly neural network layers and parts of datasets, but this should not lead the reader to think that these are the only possible ways of conducting an ablation study: many parts of a neural architecture can be removed to understand the effect they produce and, more in general, they do not even need to be neurons or layers to be considered ablation studies (for instance, the removal of loss functions would still be consider an ablation study).

If we narrow down this to the work that has been made regarding model ablation, we can link it to NAS because as a matter of fact we are trying different neural architectures and we are selecting the best one according to some optimization strategy that we define. However, we must clarify that NAS and ablation studies are not the same thing: NAS consists in searching different neural architectures and choosing the best one according to some specified metrics, while ablation studies in ML are experiments that aim to understand causality in a system by removing its components (this specific definition is inspired by a famous tweet by F. Chollet[1]).

1.6 Delimitations

The focus of this work is on ablation studies, involving mainly neural network layers and parts of datasets. Other related fields of AI optimization like HO will be mentioned, but they do not constitute the main point of the thesis, and they are presented only for the sake of completeness, to give the reader a clearer picture of the topic of the dissertation. Regarding the frameworks and platform that will be discussed, it is easy to understand that we will mostly refer to PyTorch, often compared to TensorFlow to stress how they differ in the usage and to highlight what needs to be implemented for a framework that will operate on top of them.

1.7 Outline

In the next chapters of this thesis we are going to present, first, the necessary background knowledge to understand the work, followed by the design of the ablation study framework implemented in Maggy; finally we illustrate the results obtained and the conclusions that can be drawn.

[1] `https://twitter.com/fchollet/status/1012721582148550662?lang=en`

Chapter 2

Background

In this chapter we are going to present all the necessary concepts that the reader needs to follow this dissertation. At first we give a general overview on AI and its meaning, then neural networks, *Automated Machine Learning (AutoML)* and ablation studies are explained. Lastly, we present the frameworks and platforms that have been used for this thesis work, together with the original architecture of Maggy [23], the key framework on which this work has been conducted on.

2.1 A Brief History of AI

AI has always been a topic of great interest in computer science. Its ultimate goal is to produce machines able to take decision autonomously, but we can see that the meaning of AI has muted considerably throughout the years. We will now provide a brief overview of AI from the first appearance in modern history until today.

2.1.1 What is Considered AI

Many definitions of AI have been given and, as technology advances, more and more are proposed. To look at some of them we can cite:

- John McCarthy[1], during the Dartmouth AI conference in 1956, which is considered as the one what that gave birth to AI said "The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature

[1]https://www.livinginternet.com/i/ii_ai.htm

of intelligence can in principle be so precisely described that a machine can be made to simulate it."

- The Encyclopedia Britannica[2] reports "AI, the ability of a digital computer or computer-controlled robot to perform tasks commonly associated with intelligent beings."
- The dictionary Merriem-Webster[3] defines AI as "a branch of computer science dealing with the simulation of intelligent behavior in computers" or "the capability of a machine to imitate intelligent human behavior"

According to Alan Turing, sometimes referred to as the "father of computer science", an AI agent should be able to trick a human into thinking that she is talking to another human after receiving an answer given by a computer. This process is defined by the Turing Test [39], proposed by A. Turing in 1950 and since then, many things changed. Nowadays, this test is not a popular choice to measure machine intelligence anymore, and computer scientists prefer more complex tasks such as describing pictures with correct sentences [18], object detection or classification [33], understanding human language [42] and so on. We have reached a point where it's hard to quantify the performance of such machines because sometimes not even humans agree on the same evaluation.

2.1.2 Artificial Intelligence Today

AI is becoming more and more popular in these last few years thanks to the important results that we continuously see in research: fields such as computer vision, automatic translation and many others have reached surprising results through the development of specific neural network models that outperform by far any previous approach ever attempted. Neural networks are becoming so popular despite the fact that very few people understand them deeply and, even when result seem very promising, data scientists and engineers really struggle to interpret and explain such good performances. For this reason, neural networks have been frequently referred to as "black box" models since sometimes they can be overly complicated: we do know what is happening inside them in the sense that the mathematics of these models is completely specified, but we as humans still can not interpret or explain the true

[2]`https://www.britannica.com/technology/artificial-intelligence`
[3]`https://www.merriam-webster.com/dictionary/artificial%20intelligence`

reason why machines make certain predictions. What we do is just feed a model with a large number of examples together with the output that we expect, and we hope that the model will learn the general underlying rule to link input and output. Neural networks have existed for decades, but only recently they have outperformed previous approaches thanks to large datasets, powerful hardware and new training algorithms.

2.2 Neural Networks

In this section we are going to present the basic concepts to understand neural networks, their components and how these models are trained. This section is important to understand the architecture from which we start when we define an ablation study.

2.2.1 What is a Neural Network

Currently many AI algorithms are implemented with a neural network, and the best way to explain a neural network is probably with a visual example, as it is shown in Figure 2.2.1. This specific example shows a *feed-forward* neural network because the nodes in the network do not form a cycle. A network is made of many layers, in turn made of nodes (also called artificial neurons) connected to one another in subsequent layers.

The first layer is called the input layer because every node in it receives a numeric input that will be propagated throughout the network, whereas every other node in the network takes different inputs (the outputs of the previous nodes), multiplies them by the weight that is associated to each connection, and finally sums up all together. An activation function is then applied to the output that is generated from each of these nodes. Activation functions are non-linearities introduced in the network to make it able to produce non-linear space segmentation, and therefore solve complex problems. Often, layers have a special input called “bias” which is nothing but a node in the layer whose input is always set to 1 and from a geometrical perspective, biases are needed to separate the n-dimensional problem space with hyperplanes that do not intersect the origin. The final layer (or output layer) produces the final decision of the network for a classification or a regression problem. There are many other types of neural network

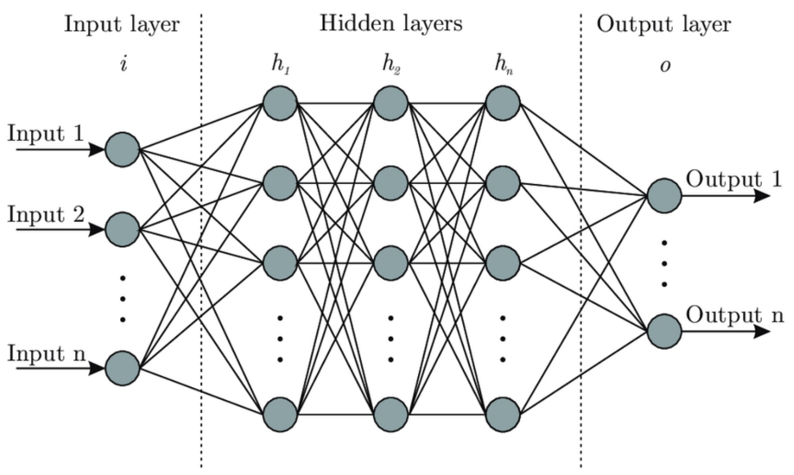

Figure 2.2.1: An example of deep neural network [27]

that we will not mention here for brevity, but the general principle remains the one that we just illustrated.

Everything that is referred to as deep learning is implemented through a neural network, even though that is not the only way for computer scientists to implement ML algorithms as we can see in the taxonomy in Figure 2.2.2. Even though neural networks are so popular nowadays they were invented in the 1950s [5] but they were hardly considered useful due to the famous *Exclusive OR (XOR)* problem (a thorough explanation of the problem is given in Chapter 6 of the Deep Learning Book [10]). Given an input of two bits, a neural network can not learn the logical XOR function without the introduction of non-linear activation functions. The problem is well understandable when seen from a geometrical point of view: if we imagine each layer as a linear transformation of the previous outputs, the final output is still a combination of this linear transformations and so it is linear itself. However, as we can see in Figure 2.2.3 a straight line can not separate our two classes in the correct way. The introduction of non-linear activation functions between layers solves this problem as it produces non-straight lines in the same space.

The XOR problem led to the first "AI winter" because it was deemed that neural networks were inadequate to solve most problems. From that point on, other AI winters followed, mainly because computers were not powerful enough to run AI models or because no model was found adequate to solve a certain task.

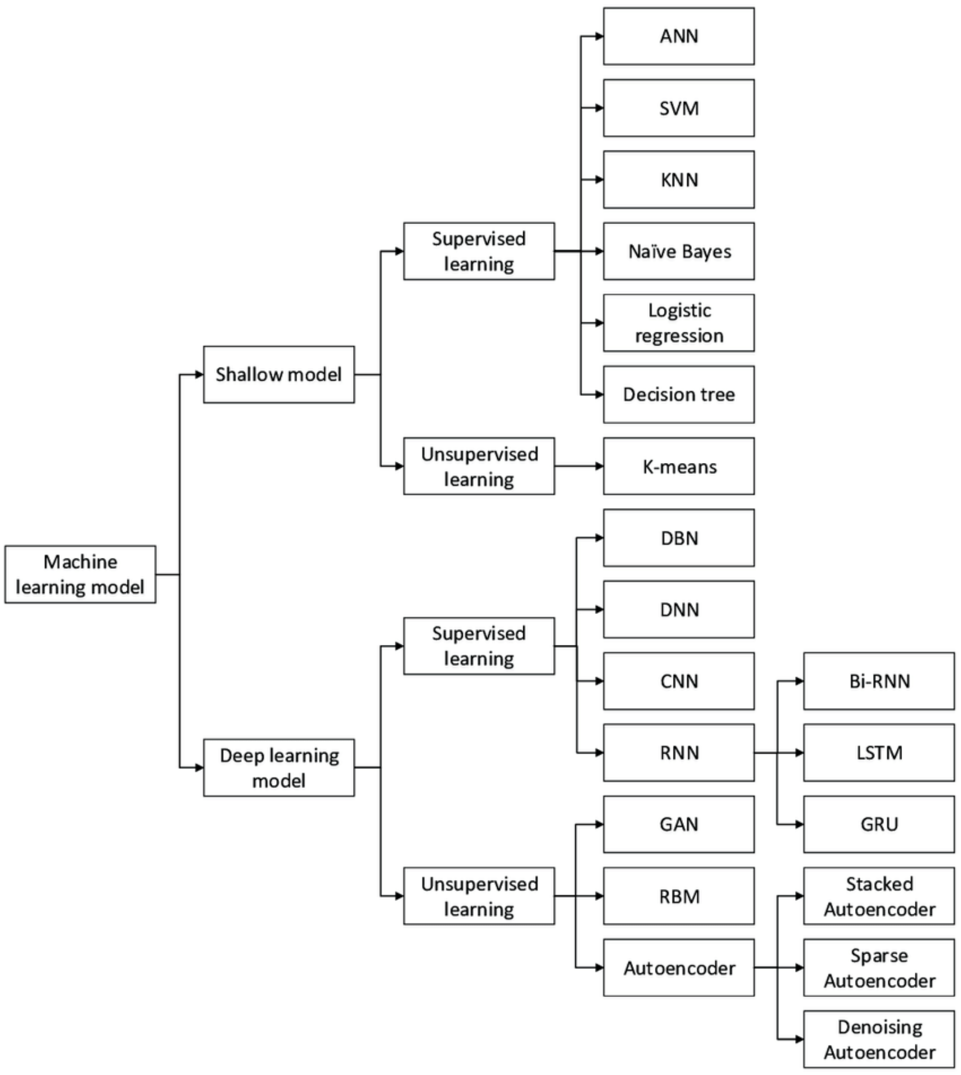

Figure 2.2.2: Machine Learning Taxonomy [20]

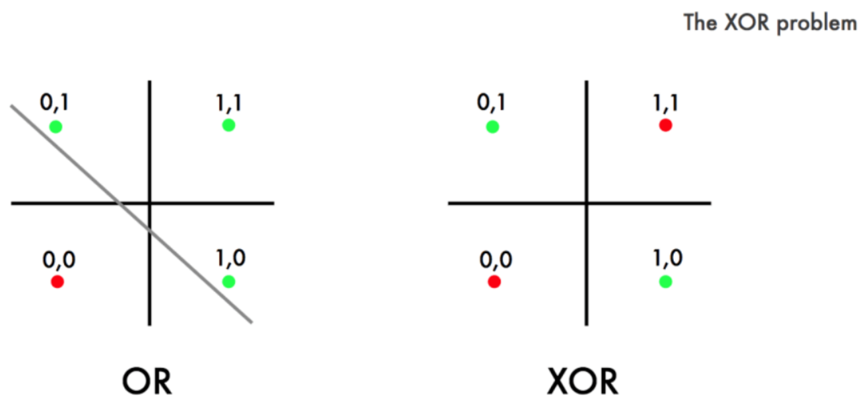

Figure 2.2.3: The XOR problem from a geometrical perspective [26]

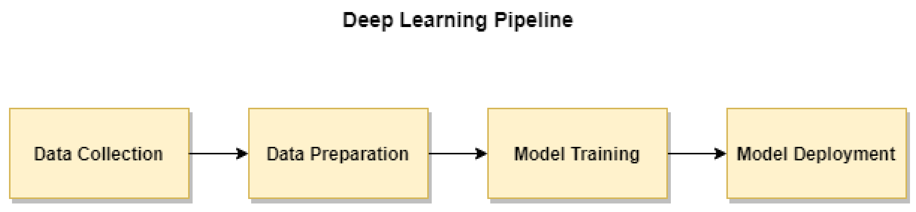

Figure 2.2.4: A simplified Deep Learning pipeline

2.2.2 Training a Model

Once a neural network model has been built, it is time to train it: this is the major difference compared to more traditional algorithms with which computer scientists have always worked. In fact, instead of instructing a machine with what to do for every possible input, we just show it a series of examples and the output that we expect, and we just hope the machine will learn the general rule in between through the minimization of a cost function that we define.

Mathematically speaking, the training of a feed-forward neural network happens with a process called backpropagation, which is nothing but the minimization of the error (that we define) between the machine prediction (i.e. the output of the final layer) and what we expect to be predicted. Backpropagation consists of a forward pass and a backward pass: the forward pass is when the data is fed into the model and this produces an outcome which is in turn used to compute a loss metric. In the backward pass, the gradient of the loss with respect to every weight is computed, and it is used to update the model weights. For this reason, model training is by far the most computationally heavy phase in the entire deep learning pipeline (we can see a simplified representation in Figure 2.2.4).

Training a network requires two main steps:

- Defining the model architecture
- Defining the model hyperparameters (i.e. all the variables that will not be learnt during training)

Both these tasks are quite difficult for humans, in the sense that people just follow best practices to choose them (e.g. *Convolutional Neural Networks (CNNs)* are used for image and video processing (such as object detection and recognition), *Recurrent Neural Networks (RNNs)* are used for *Natural Language Processing (NLP)* and time series analysis), while coming up with a new architecture or a novel strategy for hyperparameter choice is only a research topic. It is for this reason that recently two important fields in AI, namely NAS and HO, have gained more and more popularity. As these names suggest, NAS refers to the automatic search of the best model for a given problem, and HO refers to the search for the best model hyperparameters.

Currently, even though NAS and HO are often mentioned together, as they are two really close topics in ML, many complete tools for HO are already available and ready to use (to name a few we can list Optuna [2], Auto Sklearn [9], Hyperband [17], Automatic Model Tuning for Amazon SageMaker[4], etc.), while NAS tools (like Autokeras [15], ENAS [7, 8], NNI [30]) offer simple functionalities like the comparison of different ML techniques and libraries or they provide neural search limited to small components of the architecture, like the design of a single RNN cell [36]. Especially in these past few years, the research on AutoML is proceeding at a fast pace, but there is certainly still a lot to do. More information on the aforementioned tools can be found in Section 4.3.

2.3 Why an Ablation Study

Even though the term "ablation study" is now used mostly in AI, the term is taken from medicine and dates back to the XIX century. We are now going to introduce the terminology and the concepts that will be reprised throughout this thesis work.

2.3.1 Ablation Studies in the Past

The term "ablation study" is borrowed from the medical field, since originally it was a technique introduced by the French physiologist M.J.P. Flourens [25] consisting in removing parts of the nervous system of vertebrates to understand their purpose. In the early 1800s he conducted various experiments on pigeon brains and successfully proved the functions of cerebral areas like the *Cerebellum*, the *Medulla Oblungata* and some brain lobes, although only to an approximate understanding. The approach that we take with neural networks is very similar (yet, less cruel) to what Flourens did with vertebrates in the sense that we remove parts of our models to see how well the model performs without them. To a certain extent we could also argue that we

[4]`https://aws.amazon.com/sagemaker/`

try to understand the purpose of these ablated parts, even though the link between performance and purpose is often not well understood.

A prominent difference with physiology is that even though an animal brain is enormously more complex than an *Artificial Neural Network (ANN)*, it is not modifiable, while in ANNs we can move parts and layers in different positions, tune parameters and produce an infinite number of combinations with the components at our disposal. Since the problem is so complex, best practices have always helped data scientists in the quest for the right model, but there is still no evidence for a universally correct method for how to build a neural network. Maybe in the future we will have more sophisticated tools to examine such complex models, in the same way that we found new tools and methods to study our own brains in 200 years of scientific progress.

2.3.2 Ablation Studies in this Work

In this thesis we are going to talk mainly about layer ablation and feature ablation. The former consists in removing entire layers of a neural network, while the latter focuses on removing parts of training datasets (by "parts" we really mean any type of data that the user can come up with). In the version of Maggy previous to this thesis it was only possible to ablate entire columns of a dataset, and this extension is one of the major contributions of this work. We will refer to the *ablator* as the logic that defines how layers and features are removed. For instance, Maggy implements the *Leave One Component Out (LOCO)* policy, which requires the user to specify a series of components that will be removed one by one: a series of trials will be run, and for each of them only one component will be removed from the initial system (model and dataset). To produce the ablated models and datasets, Maggy uses some *generators*: these are functions that take a layer (or a feature) identifier in the system as a parameter and return the code to instantiate that same model (or dataset) without the component indicated by the identifier.

2.3.3 The Role of Ablation Studies in Research

In the academic community an ablation study is useful to validate research results before publishing them. For instance, coming up with one well-performing model could be just a lucky coincidence, whereas an ablation study allows the researchers to justify with more confidence the model choice, suggesting the necessity of some components rather than others. Unfortunately, these studies are not found so frequently in scientific literature since it takes time and energy to arrange them. To the best of our knowledge, Maggy is at the moment the only tool to devise automated ablation studies for deep learning models in Python.

2.4 The Concept of AutoML

There has been a lot of excitement about AutoML in the last years and, as it generally happens, definitions are often too broad and vague to be used with a precise meaning. In this work we are going to refer to AutoML according to the definition given in [13], where it is divided in three categories:

- HO
- Meta-learning
- NAS

HO consists in finding good values for parameters such as learning rate, number of epochs, size of convolutional filters, etc. that are normally fixed during the training process. Meta-learning means “learning to learn” as Y. Bengio et al. describe in their paper [4]. At a high-level meta-learning can be represented with two loops one inside the other, where the inner loop is the usual training loop and the outer one modifies the structure of the network itself over time. NAS aims to optimize neural architectures as the name suggests. We can further divide this field in three sub-tasks for which we have to define:

- a search space
- a search strategy
- a performance estimation strategy

Firstly we need to decide which possible architectures we are going to try (search space), then we choose how to explore these architectures (search strategy) and finally we choose how to evaluate what we have explored. In theory, there are infinite possibilities to explore new models, but in practice it’s quite common to start from a well-known and well performing architecture and apply little tweaks here and there to define a search space.

Regarding the search strategy there are two big families of methods that we can apply: directed and undirected search. The first one means that we choose the next model to explore based on the result that we obtain in other explorations, while in the second one every search is independent. Classic examples of undirected search are random search or grid search: random search picks (as the name suggests) randomly some points from the search space, while grid search is an exhaustive search through a manually specified subset of the entire search space. Finally, regarding the strategy to estimate model performance one can choose metrics such as accuracy, loss, speed or other metrics according to the final goal that is to be achieved.

Of the three branches of AutoML that we have mentioned, this thesis focuses mainly on NAS as an ablation study can modify the structure of the neural network that we are dealing with.

2.5 Frameworks and Platforms

In this section we will briefly go through the platforms and frameworks that have been used in the thesis. We will also provide a description of the original Maggy framework and its functioning.

2.5.1 Hopsworks

Hopsworks [14] is a platform developed by Logical Clocks [21] that provides a suite of tools for ML tasks, especially in a distributed environment. Originally, it started with the creation of HopsFS, a distributed *File System (FS)* underneath Hopsworks based on *Hadoop File System (HDFS)* [3] that achieves better performance dealing with both small and large files (HDFS is not optimized for small files). Hopsworks offers a series of tool to support ML from training to production and gives the user the possibility to create end-to-end pipelines comprising data ingestion, preparation and storage, as well as model training, analysis and deployment.

On top of HopsFS and together with Hopsworks, Logical Clocks created also an open-source Feature Store[5]. A feature store is a tool to ingest, store and version the data (features) of an ML system. Among the advantages that it provides we can find:

[5]`https://github.com/logicalclocks/feature-store-api`

- Ensuring consistent features in both training and serving: poor cohesion of data used in model training and serving is one of the most frequent causes for models to perform poorly. With a feature store it just takes a glance at a dashboard to see if the two data sources are comparable.
- Data versioning and time travel queries: many developers are already used to *Version Control Systems (VCSs)* to monitor the changes in a code base, but data can change too, and exactly like code it can be versioned and accessed with queries that refer to a specific point in time.
- Feature data validation: this means that we can validate data points during ingestion before adding them to one or more features. This is necessary as poor-quality data will most certainly reduce the quality of a model built on top of them.

We will briefly demonstrate how to use the Hopsworks Feature Store in the context of ablation studies in Chapter 3.

2.5.2 PyTorch

In the past few years, TensorFlow [1] with its API Keras [6] has been the most popular choice for data scientists to build deep learning models. The reason is that Keras provides an intuitive an easy-to-use interface while TensorFlow provides the core functionalities for the execution. However, PyTorch [35] has recently gained much popularity as it can be seen in Figure 1.2.1 for some of its most interesting features, which are:

- Being Pythonic: data are represented as tensors and they provide an interface really similar to NumPy [40], datasets and models are Python class, the training function is a simple Python function (debugging a training function in TensorFlow is not immediate, while in PyTorch a standard debugger would work).
- Being researchers-oriented: the simplicity as well as the low-level control that PyTorch gives to its users is one of the main reasons for its spike in popularity in academia as we saw in Section 1.2.
- High performance: especially PyTorch implementation splits control flow and data flow, leveraging CUDA [32] to optimize the computation among different GPUs and an relying on optimized C++ code executed on the *Central Processing*

Unit (CPU).

2.5.3 Apache Spark

Apache Spark [34] is a unified analytics engine for large-scale data processing that offers APIs in Scala, Python, Java and *Structured Query Language (SQL)*. Spark saves intermediate results in memory and it optimizes its pipelines before every execution. It was created to handle really large datasets and therefore is a very popular tool in the field of Big Data. Two of the main data structures that it provides are Dataframes and *Resilient Distributed Datasets (RDDs)*: these are both implemented in a way that allows data to be stored and processed in parallel across multiple nodes of a cluster, resulting in a much faster computation compared to traditional data analysis frameworks (such as Pandas[6]). In Maggy, Spark is used as back-end for distributed computing: a Spark driver keeps information on the whole experiment (which might be an ablation study or HO) and the various trials run in parallel on different Spark executors, eventually reporting back to the driver. We illustrate this mechanism more in detail in Section 2.6.

2.6 Maggy

We are now going to show a high-level overview of the architecture of Maggy. Except for the component *Maggy API* this architecture has not been modified, and constitutes a previous work by M. Meister and S. Sheikholeslami presented in [28].

In Figure 2.6.1 we can see four main components:

- the Maggy API
- the Spark Driver
- the Spark Executor
- the Distributed FS (which in the scope of the thesis is HopsFS)

The Maggy API component is the only one that is exposed to the end user, and it supports both ablation studies and HO experiments. Within ablation studies we can distinguish modules for TensorFlow (already existing previous to this thesis work)

[6]`https://pandas.pydata.org/`

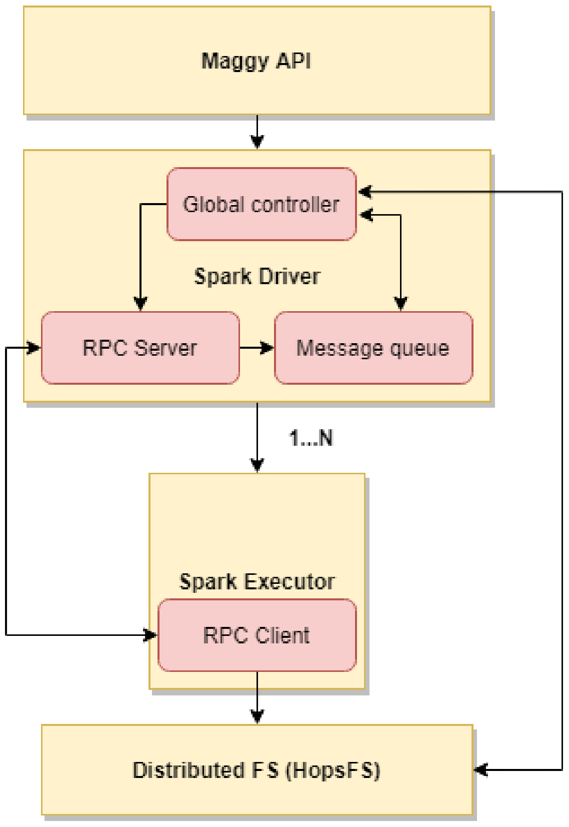

Figure 2.6.1: Architectural overview of Maggy

and PyTorch (main scope of the thesis). Even though Maggy is framework-agnostic (i.e. allows to run experiments without imposing a single deep learning framework), the support for TensorFlow and PyTorch reduces considerably the complexity for the final user. Without this support, the only way to define new experiments would be to write the code for the whole deep learning pipeline for every configuration that one wants to run, and this would be obviously really inefficient. Some of the contributions of this thesis work are the support for ablation studies in PyTorch, as well as the simplification of the developer API by separating the PyTorch-specific ablation logic from the logic regarding generic Maggy functioning (it is in fact possible for developers to write custom ablators that best suit their needs).

Since an API optimized for TensorFlow was already present when I started this thesis work, I kept the original API unaltered, to allow a seamless transition for users that want to test models in both PyTorch and TensorFlow. The other components in the graph underlying the API purely concern back-end and have not been modified in this thesis work, but we will mention them anyway for completeness's sake.

First of all, we need to specify the meaning of *trial*. Maggy is a framework that provides functionalities for asynchronous execution of ablation studies and HO experiments. We will refer to a trial as the execution of the code for either HO or ablation on a Spark executor. Since generally many trials are executed in parallel in a single experiment, these trials are stored in a queue waiting to be run and they are represented as Python dictionaries. As soon as a Spark executor is idle, it will poll the queue, taking the necessary information to run that trial and finally it will report the results. The code to run the trial is specified by the user before starting the experiment, as we can see in Section 3.4

The Spark driver takes care of scheduling trials and manages the *Remote Procedure Call (RPC)* Server thread, which communicates with the RPC client in the Spark executor, periodically receiving trial metrics and status messages. This information is later passed onto the message queue. The *Global Controller* polls the trials queue, checks if trials need to be early-stopped, gets new trials and assigns them to the Spark executors (only one is represented in the figure, but of course a classic Spark configuration supports multiple executors).

On the Spark executors, the actual trials are run until they are successfully completed or early-stopped. If an executor is not running any trial, it queries the Spark driver to obtain another one to execute until there are no more trials in the queue. The RPC client communicates the trial outcome to the RPC server. Maggy supports early-stopping only for HO through median rule, this means that it stops a trial if it is performing worse than the median of all the metrics of the trials previously executed. Finally, the *Distributed FS* is used as a persistent storage layer, essentially used to retrieve the dataset used to run the experiments and to store the obtained results.

Chapter 3

Design and Implementation

In this chapter we are going to discuss what actually has been developed to provide support for ablation studies in PyTorch. First, we will present the design principles that have been followed through the whole project, and then we show how layer ablation and feature ablation have been designed and implemented. We also provide the code to run both layer and feature ablation experiments with Maggy and finally we propose an alternative API that has not been implemented in Maggy but that can still be considered interesting for the extension of the framework.

3.1 Design Principles

Developing a framework that will be used by other developers is not an easy process. Throughout the project, three key points were set as guidelines:

1. Efficiency: training a model is a lengthy and computationally-heavy task and therefore it is necessary where possible to save compute and time, which increase linearly with the number of trials that the user wants to run. Maggy supports early-stopping of trials that perform worse than the median of the metrics of the previously executed trials and in this way it can save resources that can be allocated for more promising trials.

2. Ease of use: a necessary condition for a framework to be largely adopted is its ease of use. I tried to keep the interface really simple, with a few classes exposing only the necessary methods that should be straightforward and usable with just a glance at the documentation. Let us remember that most ablation studies are

currently carried out manually, and people will continue to do so if the framework that we propose is complicated to use.

3. Minimal code changes (compared to normal PyTorch code): an important goal for this project was to minimize the additional code - compared to the initial PyTorch code without ablations - that one needs to write in order to use Maggy. Since PyTorch is already quite a verbose framework, and since ablation studies are useful but require a lot of code changes (when conducted manually), one of the main goals of this framework is to allow developers to perform ablation studies writing only a few more lines of code.

3.2 Implementation of Layer Ablation

Ablation studies in Maggy consist of two main branches: layer ablation and feature ablation. The former regards the removal of layers from a neural network, while the latter focuses on the removal of features intended as parts of the training dataset. These features might be, for instance, simple columns in a tabular dataset, channels in an image dataset or even words in a text corpus. We will now illustrate how layer ablation has been implemented as well as the main challenges along the path.

3.2.1 LOCO Ablator

An *ablator* here is defined as the logic implemented to remove the desired components from a neural architecture. As we are going to show in this chapter, the Maggy API requires first the users to specify the components to be removed and then the *way* in which they will be removed (in other words, the ablator). At the moment, the only ablator that Maggy supports is LOCO, but others can be defines through the developer API. LOCO is a simple ablation policy that consists in removing a single component from a system, and in case of a feed-forward neural network this means removing a single layer. Stacking more layers in a neural network is considered an easy best-practice to increase the accuracy of the model; the downside is that the complexity of the model increases accordingly. Large models are slower both during training and inference, and sometimes it is not so easy to understand what is a good trade-off between complexity and accuracy. By removing unnecessary layers and collecting relevant metrics, Maggy can help find this desired trade-off. A visual example of how

LOCO works is displayed in Figure 3.2.1.

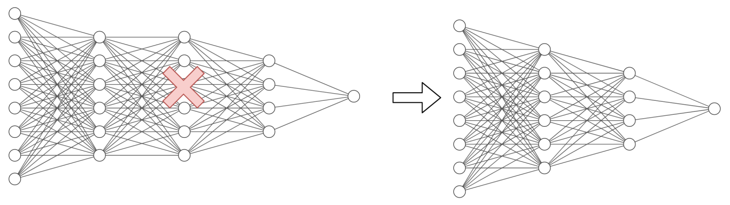

Figure 3.2.1: LOCO policy: layer ablation

The implementation of the ablation logic can be resumed in four steps:

1. Getting the list of the model's modules: modules are defined as the single parts forming a PyTorch model (in the scope of this thesis they are often Linear layers, Convolutional layers, Dropout layers, etc.) and in this phase it is necessary to retrieve them in a form of Python list that we will later use to edit the shape of the overall model.

2. Removing the necessary modules: here the real ablation is performed, in fact this is the phase when the modules that must be ablated are actually removed from the model (which is still represented as a Python list of different modules).

3. Matching the modules' attributes: after the ablation we need to restore the network because modules in a Sequential PyTorch model are arranged in such a way that by removing a module we will probably not be able to use our model anymore. This is due to the fact that there are some conditions that must be respected between modules in the neural network; for instance, when there are two subsequent Linear layers in the model, the attribute *output_features* of the first one must match the attribute *input_features* of the second one. Since this is an important point of this work we will elaborate on that in Section 3.2.2.

4. Instantiating the new model with the correct attributes: finally, when we restored the model after the ablation we still have the single modules in a Python list, hence we need to instantiate again the whole PyTorch model to be able to use it again for training and inference.

These steps are executed every time Maggy runs an ablation trial. We also need to specify that a new couple model/dataset is generated at every trial through some functions that we defined as *generators*. The functions take as parameter the identifier of the component to ablate in the system (model or dataset) and return the code that is needed to instantiate the ablated system. In this way we do not have to take care of restoring the original system after the ablation trials have been completed because that is never altered. In Section 3.4.2 we are going to discuss more in depth the role of generators.

3.2.2 Shape Inference

Shape inference was the main challenge in the implementation of layer ablation: to describe this problem we need to take a step back and analyze the difference between the APIs to build neural networks that PyTorch and TensorFlow provide. In Listing 3.1, taken from the official website, we can see that TensorFlow provides a really straightforward method to instantiate layers and models: for Dense layers we just need to specify how many neurons we want in them and, additionally, we can attach an activation function (e.g. 'relu') to a layer. In general, we need to specify the input shape of the data in the first layer of the network.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10)
])
```

Listing 3.1: Model instance in TensorFlow

Taking a look at PyTorch's API in Listing 3.2 we can see a few differences compared to TensorFlow:

- For each Linear layer (equivalent of Dense in TensorFlow) we need to specify both input and output features, which do not refer to nodes in a layer but to the size of the tensor before and after passing through the layer. In other words, we must specify the connections between two layers, instead of the number of nodes of the layers themselves. In this case the input tensor is an image of 28×28 pixels that after flattening becomes a 784-dimensional vector (note that we do not have to specify the vector dimension after flattening in TensorFlow), and after passing through two Linear layers we get a resulting vector of 10 elements. Eventually, the instantiation of the Linear layer is nothing but a matrix of shape (*output_features*, *input_features*) that is multiplied by the input tensor.

- There is no specification of the input shape in the model. Even if in some cases it can be inferred from the first layer, we will see that this cause some problems during feature ablation. In fact, if we modify the structure of our input data there is no general rule to modify the first layer accordingly, unless the user specifies it.

- Activation functions are not part of layers but they are separate objects not linked to anything else in the model. This causes some complication during layer ablation, as it would be natural to think that when we remove a layer we remove also the activation function associated with it.

```
model = torch.nn.Sequential(
    torch.nn.Flatten()
    torch.nn.Linear(in_features=784, out_features=128),
    torch.nn.ReLU(),
    torch.nn.Linear(in_features=128, out_features=10),
)
```

Listing 3.2: Model instance in PyTorch

This problem has been solved creating a random tensor of the input shape that the model requires, and then passing it through the model layers, tracking its shape step by step. If PyTorch accepts the tensor it means no changes must be applied, otherwise, if the tensor is not compatible with a certain layer, Maggy will take care of resizing the network accordingly to allow the tensor to pass through it. Let us make a visual example to clarify this.

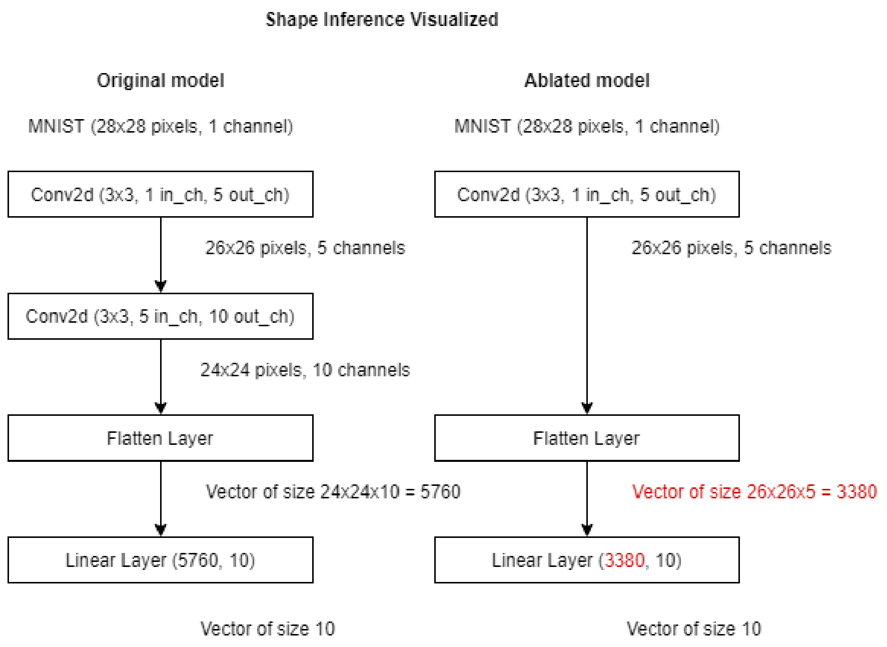
Shape Inference Visualized
Original model
MNIST (28x28 pixels, 1 channel)
Conv2d (3x3, 1 in_ch, 5 out_ch)
26x26 pixels, 5 channels
Conv2d (3x3, 5 in_ch, 10 out_ch)
24x24 pixels, 10 channels
Flatten Layer
Vector of size 24x24x10 = 5760
Linear Layer (5760, 10)
Vector of size 10
Ablated model
MNIST (28x28 pixels, 1 channel)
Conv2d (3x3, 1 in_ch, 5 out_ch)
26x26 pixels, 5 channels
Flatten Layer
Vector of size 26x26x5 = 3380
Linear Layer (3380, 10)
Vector of size 10

Figure 3.2.2: The illustrated shape inference problem in PyTorch

In Figure 3.2.2 we can see an example of a very simple CNN with the MNIST dataset, a famous dataset containing hand-written digits that range from 0 to 9. The original data are tensors of shape $1 \times 28 \times 28$ (one channel, 28×28 pixels each image). Passing through the first layer we lose 2 pixels on each dimension (because the convolutional filter is 3×3) and we get 5 channels as we specified 5 out channels in the first layer. Following the same reasoning, passing through the second layer the tensor becomes of shape $10 \times 24 \times 24$ and finally, after flattening, all the features in the tensor are aligned in a vector. The size of this vector is $24 \times 24 \times 10 = 5760$, so if we want to add a linear layer after the flattening one (as it is a common step for classification) we need a layer with 5760 features. Now, let us say that we want to ablate the second layer because we are not sure if that really contributes to the accuracy of our model: how many input features do we need in the Linear layer now? There is quite some math to do, which is not too difficult for this specific model, but chances are that the user will have a far more complex model and she will want to specify multiple ablations, and we can see that it quickly becomes a lot of work. Now Maggy takes care of this automatically before the actual training function is called, specifically it passes a random tensor of the shape that the user specifies through the model and, if an ValueError is raised, it can see the last valid shape of the tensor before the error, and reistantiate the layer that caused the exception with the correct parameters.

Someone might argue that the specification of the input shape is not necessary as it can be inferred from the dataset of our ML task. Unfortunately, unlike TensorFlow with TFRecords (where data are quite structured) PyTorch Datasets are extremely flexible and have absolutely no specification on the data they contain. In this way a PyTorch Dataset could contain, for instance, images in a tensor of shape (*image_height* × *image_width* × *channels*) as well as (*channels* × *image_height* × *image_width*). Even though the first form is more frequently used, both are valid conventions and the framework should not impose one over the other. For this reason, if the user wants to ablate the first layer of a model, Maggy has no way to know what will be the new input shape of the model after the feature ablation has been executed, unless the user herself specifies it.

3.3 Implementation of Feature Ablation

As we mentioned previously, feature ablation concerns the removal of parts of a dataset. Previous to this thesis work, Maggy supported only the ablation of dataset columns, but we are going to discuss how this concept can be generalized and extended to the ablation of any data type. Also for feature ablation, LOCO is the only supported ablator for now, but the user can decide to implement her own custom ablator through the developer API. Here in Figure 3.3.1 is a visual example of the LOCO policy behavior.

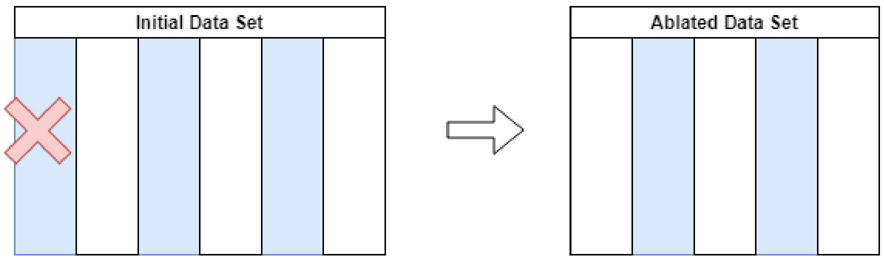

Figure 3.3.1: LOCO policy: example of feature ablation

The next three sections are going to describe the implementation of the LOCO policy for feature ablation. First, it was necessary to come up with a way to identify the data to include in the ablation study since PyTorch provides a very flexible interface to store datasets as we will see in Section 3.3.1. Once the data is identified, we can proceed in two directions: we can harness either the class *Dataset* or the class *DataLoader* for the ablation: since PyTorch defines both to handle data, Section 3.3.2 is an analysis of which one we should use to perform feature ablation. Finally, in Section 3.3.3 we observe that feature ablation involves also the modification of the model architecture, so we describe in what way the model has to be changed to be compatible with the ablated dataset.

3.3.1 The Fixed Points of PyTorch

The design of this framework has been particularly challenging because of the great flexibility that PyTorch allows. If we look at the Dataset class defined in *torch.utils.data* (Listing 3.3) we can see that the code is really simple and it provides the minimum support for everything to work smoothly, and at the same time it lets the user code most other functionalities.

```python
class Dataset(object):

    def __getitem__(self, index):
        raise NotImplementedError

    def __add__(self, other):
        return ConcatDataset([self, other])
```

Listing 3.3: Dataset class implementation in PyTorch

The only thing that is necessary to implement a PyTorch Dataset is to subclass *torch.utils.data.Dataset* and overload its *__getitem__* method. Additionally, the user can also overload the method *__len__* but this is not compulsory. There is no specification on where the actual data should be stored, nor in what format it should be returned when calling *__getitem__*, nor any other metadata that are generally useful for a dataset class. It is perfectly fine for PyTorch if the user instantiates a Dataset and stores the data in *self.data*, or *self.table.data*, or any other attribute inside or outside the class.

The *__getitem__* method will be called inside the training function (still completely specified by the user), in turn called when iterating over another PyTorch class (the Dataloader) to feed the data into the model. Again, to give an example of the flexibility of the framework we can say that *__getitem__* can return either objects like tuples in the form (*label*, *data*), Numpy [40] arrays, PIL images or anything that the user can think of. Since the DataLoader is called inside the training function defined by the user, the data can still be processed inside such function before being fed into the model (the code in Listing 3.4 gives a practical example of this).

If we compare training functions in PyTorch and TensorFlow, as we can see in Listing 3.4, TensorFlow generally requires one line of code for the training while PyTorch requires a lot more. This is one of PyTorch's most appreciated features as it allows users a lot more control on what is happening and if some things do not go as expected this function is easy to debug. On the flip side, there is a lot more code to write, and some concepts (like accuracy) are not even defined by the framework.

```python
# Implementation of a training function in PyTorch
#  (taken from the official PyTorch documentation https://pytorch.org/
    tutorials/beginner/blitz/cifar10_tutorial.html)

for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')

```

```
# Implementation of a training function in TensorFlow
#  (taken from the official TensorFlow documentation, https://www.
    tensorflow.org/api_docs/python/tf/keras/Model)

model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
```

Listing 3.4: Comparison of training functions, in PyTorch and TensorFlow

This flexibility is key for the design choices in Maggy. Following the same philosophy I have decided to implement feature ablation by letting the user subclass *MaggyDataset*: an abstract class containing only one method to implement (*ablate_feature*), as we can see in Listing 3.5. The method *ablate_feature* takes a feature as parameter and should specify how the dataset class should remove such feature from where the data is stored. There is no need to restore the dataset to its original version because Maggy generates a new ablated dataset every time that a trial is executed. In this way we can keep feature ablation simple, elegant and flexible: it is implemented with a few lines of code, it respects PyTorch philosophy and it allows the user to ablate any type of data (not just columns of a tabular dataset but also channels of images, parts of time series and, in general, anything that the user can come up with). Furthermore, with this implementation, Maggy does not need to know where the data is stored or how it is retrieved by the DataLoader, as the user will code the *ablate_feature* method according to the way that she had previously coded the Dataset class. We will see some possible applications of this in Chapter 4.

```
from torch.utils.data import Dataset
class MaggyDataset(Dataset):

    def ablate_feature(self, feature):
        raise NotImplementedError
```

Listing 3.5: Class MaggyDataset

Even though this solution covers as a matter of fact every possible feature ablation, it is also true that the user has to specify how these ablations will occur by overriding the *ablate_feature* method. Since there might be some recurrent types of dataset across many data science projects, it makes sense for the framework to hide some of the complexity to the user by defining some more classes that can deal with predefined data structures. For instance, a typical dataset that most data scientists come across is the Pandas DataFrame[1], and that is why we decided to include in Maggy a class that can handle it, defining all the methods and attributes that the user herself should otherwise code. In Listing 3.6 we can see how this class, called PandasDataset, is implemented.

```python
# This class subclasses torch.utils.data.Dataset
class PandasDataset(Dataset):
    def __init__(self, df, label):
        super(Dataset).__init__()
        self.label = label
        self.columns = df.columns.values.tolist()

        if self.label not in self.columns:
            raise ValueError(
                "The specified label can't be found among the dataset columns"
            )

        label_idx = self.columns.index(label)
        self.columns.pop(label_idx)
        self.data = np.delete(df.values, obj=label_idx, axis=1)
        self.labels = df.values[:, label_idx]

    def __getitem__(self, item):
        return self.labels[item], self.data[item]

    def __len__(self):
        return self.data.shape[0]

    def ablate_feature(self, feature):
        lowercase_columns = [col.lower() for col in self.columns]
        feature = feature.lower()
        if feature.lower() in lowercase_columns:
            print("Ablating feature {}".format(feature))
            idx = lowercase_columns.index(feature)
            self.data = np.delete(self.data, obj=idx, axis=1)
        else:
            raise RuntimeError("Ablation failed: column not found")
```

Listing 3.6: Class PandasDataset in Maggy

[1]https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

3.3.2 Ablation of Dataset or DataLoader

As we already mentioned in Section 3.3.1, PyTorch defines two classes for data handling: Dataset and DataLoader. The Dataset generally contains the actual data plus some other metadata that the user can include, while the DataLoader is the class that is called inside the training loop which splits the dataset into batches and feeds them to the neural network.

There are two points in which feature ablation can happen:

1. Where the data is stored (the class Dataset)
2. Where the data is retrieved (the method *__getitem__* of the class DataLoader)

In the first case we remove the data from where it is supposed to be stored (i.e. the class Dataset), while in the second case we let the user specify a different *__getitem__* method (in the class DataLoader) for every possible type of ablation, so only the way in which the data is returned changes, while the dataset is never altered. This second solution has eventually been discarded as it is conceptually more complex and because the class DataLoader takes some useful parameters for the training phase (like *batch_size*, *shuffle*, *num_workers*, etc.) therefore it needs to be fully accessible by the user, and Maggy should not interfere with it. We can see how the API would look like implementing the two different solutions in Listing 3.7.

```
# Option 1: pass the dataset [Implemented]

def training_fn(dataset_function, model_function):
    model = model_function()
    dataset = dataset_function()

    from torch.utils.data import DataLoader
    dataloader = DataLoader(dataset, batch_size=64, shuffle=False)

    for (batch_id, data) in enumerate(dataloader):
        # Training loop

    return loss

# Option 2: pass the dataloader [Discarded]
from maggy.ablation import AblationStudy
```

```

dataloder_kwargs = {"batch_size": 64, "shuffle": False}
ablation_study = AblationStudy('titanic_train_dataset',
    training_dataset_version=1, label_name='survived', dataloader_kwargs=
    dataloader_kwargs)

def training_fn(dataloader_function, model_function):
    model = model_function()
    dataloader = dataloader_function()

    for (batch_id, data) in enumerate(dataloader):
        # Training loop

    return loss
```

Listing 3.7: Two possible options to handle PyTorch DataLoader with Maggy

3.3.3 Impact of Feature Ablation on the Input Shape

Model ablation and feature ablation are not independent: the first layer of a PyTorch model contains information on the shape of the input tensor, and therefore it must be changed if the input tensor changes. With feature ablation we remove some components from the dataset, and this has repercussions on the model architecture. Even though different layers have different attributes that define the tensors that they accept, it is quite common for them to have an *input attribute* and an *output attribute*: for instance, Linear[2] layers have *in_features* and *out_features* (dimension of the tensor before and after the Linear layer), while Conv2d[3] layers have *in_channels* and *out_channels* (number of channels of the tensor before and after the Conv2d layer). So, if the first layer of our neural network is a Linear with *in_features* equals to 10, the network expects 10 input features, and if we ablate one of them we will need to reshape that layer accordingly.

Even though we can define some rules for simple recurrent cases (e.g. "Ablate a dataset column" $\implies$ "Reduce the input features of the first layer of one unit"), for most of the user-defined ablations we can not predict how the first layer of the neural network should be reshaped. For this reason, in order to preserve simplicity and flexibility, the best solution is to provide both an interface for data types that are most frequently used

[2]https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear
[3]https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d

(in Maggy such interface has been implemented for Pandas DataFrames, as seen in Listing 3.6) and an interface for custom feature ablation that requires the specification for the input shape of the neural network for every trial (Listing 3.5).

The API that has been implemented for Pandas DataFrames implements the same functionalities that Maggy already offered for LOCO ablation with TFRecords[4]. Section 3.4 is a comprehensive view of how to use this interface, but in general what needs to be done is passing a Pandas DataFrame to the class PandasDataset, together with the label of the predictor (i.e. the name of the column that we want to predict with our model) and the framework will take care of the followings things:

- It will automatically define the methods necessary to run PyTorch code (__*getitem*__ and __*len*__).
- It will define an ablation function that is called when the ablation study is run.
- It will store the dataset schema (otherwise this should be done manually in PyTorch).

In this way, when the feature ablation trials are run there will be no need for the user to specify the input shape of the first layer because Maggy knows already how to reshape the model if we are dealing with Pandas DataFrames. For the reasons that we explained in this section, even the order in which the dataset and model generator are called is important: in case both layer ablation and feature ablation should be executed in the same trial, Maggy must generate the model after the dataset because, as we pointed out, it needs to know the shape of the dataset to set the right parameters for the first layer of the model.

3.4 Maggy API

In this section we will discuss the general functioning of the Maggy API with an example that includes both feature and layer ablation. We will cover the definition of the experiment, its execution, as well as model and dataset generators.

[4]https://www.tensorflow.org/api_docs/python/tf/data/TFRecordDataset

3.4.1 Definition of the Ablation Study

We will now take a look at the interface with which we can define ablation studies in Maggy. We start from the idea that the most ablation studies involve the deletion of model layers or columns of a tabular dataset. Within Hopsworks Feature Store (see Section 2.5.1) the user just needs to specify the name and the version of the dataset that she wants to use, together with the name of the predictor column. In Listing 3.8 we can see an example of how to define an ablation study in Maggy: for this experiment we chose the famous Titanic dataset[5] (which contains data on a few hundred passengers of the Titanic, like age, sex, type of cabin and ticket, etc.) and it is used to train a ML model to predict if these passenger survived the shipwreck.

```
from maggy.ablation import AblationStudy

ablation_study = AblationStudy('titanic_train_dataset',
    training_dataset_version=1, label_name='survived')
```

Listing 3.8: Instantiate an Ablation Study in Maggy

3.4.2 Model and Dataset Generator

As a second step we need to specify the model on which the ablation study will be performed. The model should be defined in a wrapper function (in the example it is called *base_model_generator*) that returns the model itself. This is done in the purpose of simplicity (only a few lines of code to add to your model) and flexibility (the wrapper function does not impose any constraint on how the model should be defined). Finally, we pass the function that we have just defined to the ablation study instance through *set_base_model_generator*. Here is a simple example of what we just discussed in Listing 3.9. As a matter of fact, in this listing we just need three more lines of code (lines 1 and 14 for the function wrapper and line 16 to pass the model to the class ablation_study) compared to normal PyTorch code.

```
1 def base_model_generator():
2     import torch.nn as nn
3     model = nn.Sequential(nn.Linear(6, 64),
4                           nn.ReLU(),
```

[5](https://www.kaggle.com/c/titanic

```
                      nn.Linear(64, 64),
                      nn.ReLU(),
                      nn.Linear(64, 32),
                      nn.ReLU(),
                      nn.Linear(32, 2),
                      nn.ReLU(),
                      nn.Linear(2, 1),
                      nn.Sigmoid()
                      )
    return model

ablation_study.model.set_base_model_generator(base_model_generator)
```

Listing 3.9: Defining the base model generator in Maggy

After the definition of the base model it is time to let Maggy know what trials we want to execute. As we can see in Listing 3.10 this is quite straightforward: in line 1 we tell Maggy to remove singularly layer 2 and layer 4, but we can also choose to remove multiple layers in a single trial as it is reported in line 2. Finally, to ablate dataset features we follow the same procedure that we just mentioned, only indicating the name of the feature that we want to remove, as it is shown in line 3.

```
1 ablation_study.model.layers.include(2, 4)
2 ablation_study.model.layers.include_groups([2, 3], [4, 5])
3 ablation_study.model.features.include("Fare")
```

Listing 3.10: Defining the ablation trials in Maggy

Maggy stores its trials in the form of Python dictionaries containing all the necessary information to execute the ablation studies (or HO experiments). Each of these dictionaries stores two important functions: *get_model_generator* and *get_dataset_generator*. The former is a function that takes a parameter *layer_identifier* that returns a function containing the code to instantiate a model without the layer indicated by *layer_identifier*. An analogous process happens with *get_dataset_generator* that returns the code to instantiate a dataset without the feature indicated by the parameter *ablated_feature*. These functions are called to generate a new couple model-dataset every time that a new trial is executed.

3.4.3 Launching the Experiment

The last step of the ablation study is to wrap the training code for the model in a Python function and launch the ablation study as we can see in Listing 3.11.

```python
from maggy import experiment

def training_fn(dataset_function, model_function):
    model = model_function()
    dataset = dataset_function()

    from torch.utils.data import DataLoader
    dataloader = DataLoader(dataset, batch_size=64, shuffle=False)

    criterion = nn.BCELoss()

    optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
    epochs = 3

    for epoch in range(epochs):

        epoch_loss = 0

        for i, (labels, inputs) in enumerate(dataloader):
            # print("load number {}".format(i))

            inputs, labels = Variable(inputs), Variable(labels)
            inputs = inputs.float()
            labels = labels.float().view(-1, 1)

            optimizer.zero_grad()
            y_pred = model(inputs)

            loss = criterion(y_pred, labels)
            epoch_loss += loss

            loss.backward()
            optimizer.step()
    print("Training complete\n")
    return loss

```

```
result = experiment.lagom(map_fun=training_fn,
                          experiment_type='ablation',
                          ablation_study=ablation_study,
                          ablator='loco_pytorch',
                          name='Titanic-LOCO',
                          direction="min"
                          )
```

Listing 3.11: Wrap the training function and launch the ablation study

It is worth spending a few words on this listing to illustrate the key points. Compared to plain PyTorch code, the training function has only a few differences: the model and the dataset are taken in lines 4 and 5 from the functions that we have previously defined and, in addition, there is an object returned by the training function which is the metric needed to evaluate the trials. In this case it is the training loss, but it could also be the accuracy or any other metric that we want to maximize or minimize. Just bear in mind that PyTorch does not define accuracy by default (only losses are defined in PyTorch), so the user needs to create a custom metric if she wants to track training accuracy.

To launch the experiment we call the method *lagom* (*lagom* is a Swedish word that means “just the right amount”) of our experiment instance. The LOCO ablator will execute all the trials that we have defined, plus a trial with only the base model and dataset (no ablation whatsoever), measure the metric that we are tracking and returning the best configuration for our experiment . In this case we need to tell Maggy to minimize the metric returned by the training function because we are tracking the training loss. Here in Listing 3.12 we can see an example of the results returned after running the ablation experiment.

```
>>> Started Maggy Experiment: Titanic-LOCO, application_1591955574705_0098,
    run 1

------ LOCOPyTorch Results ------
BEST Config Excludes {"ablated_feature": "sibsp", "ablated_layer": "None"}
   -- metric 0.5779947638511658
WORST Config Excludes {"ablated_feature": "None", "ablated_layer": "[4, 5]"
   } -- metric 0.7479431629180908

AVERAGE metric -- 0.6663087904453278
Total Job Time 0 hours, 0 minutes, 51 seconds
```

```

Finished Experiment
```

Listing 3.12: Results returned after an ablation experiment

3.5 Designing the API: Simplicity or Flexibility

Extreme flexibility is appealing, but sometimes it can lead to a lot of extra work for the programmers, as we have illustrated in Listing 3.4. Another requirement of this framework was also respecting the original interface that Maggy already provided before this thesis work (i.e. the API that we described in Listing 3.10). In Listing 3.13 is reported an alternative user API for Maggy that I came up with during my thesis work. Despite this interface was not implemented in the framework in order to preserve the original API, it still has some interesting points that are worth being discussed.

- This API is easily extensible. The method *new_trial* requires the list of model layers and dataset features to be ablated, but it can take additional parameters: for instance, *infer_ablation* is used to remove together with the ablated layers also the subsequent activation functions. Adding more parameters to the method *new_trial* can change the ablator behaviour, hence there would be no need to change the interface every time that the ablation policy changes.
- Model and dataset are inherently linked together. This is an advantage as the deep learning community is going towards a more tightly coupling of model and dataset as, for instance, F. Chollet stated in his conference at ScaledML 2020 [37], saying that pre-processing layers will be soon available as part of Keras models. This is due to the fact that many models score high accuracy during the training phase, but then perform poorly when serving, often because data pre-processing is neglected in the deep learning pipeline, and the data in input during serving is not what the model (and the developer) would expect.
- Everything in one class: there is only one class (Ablator) that is exposed to the user together with three methods (the constructor, a method to create new trials and a method to execute them).

```
ablator = Ablator(model, dataset, training_function)

```

```python
ablator.new_trial((6,), ablated_layers=None, ablated_features=None)
ablator.new_trial((5,), None, "Sex")
ablator.new_trial((6,), 2, None, infer_activation=True)
ablator.new_trial((5,), 4, "Pclass")
ablator.new_trial((6,), [3, 4, 5, 6], None)
ablator.execute_trials()
```

Listing 3.13: An alternative user API for Maggy

For these reason I believe that Maggy could provide an API similar to the one presented here as it makes the usage more intuitive and simple compared to the original user interface, presented in Section 3.4. However, as I mentioned before, one of the goals of this thesis was also to preserve the original API to allow a seamless switch between different deep learning frameworks (PyTorch and TensorFlow) and for this reason the solution presented here has not been implemented in Maggy.

3.6 Extensibility

Another feature that I kept in mind during the design of the interface was extensibility. Since the project is open-source, other programmers will hopefully contribute to it, and they should have the possibility to extend it in a way that suit their needs. From an easy developer API you can get more engagement from the community, and therefore a continuously improved tool.

When I had to replicate the LOCO ablation policy for PyTorch starting from the TensorFlow module, I realized that all the logic for the ablation was in a single Python file. Of course, this prevents the developers to write *Don't Repeat Yourself (DRY)* code when a different ablation policy is to be implemented. I paid close attention to the separation of the function related to the ablation in the PyTorch framework (all placed in a module called *pytorch_ablator.py*) and the ones related to the specific functioning of Maggy, so that a new developer can just call a few functions from *pytorch_ablator.py* if she wants to create a new ablation policy.

The specific logic that concerns ablation in PyTorch involve functionalities like layer deletion in a Sequential model, matching features in subsequent layers (the problem that we discussed in Section 3.2.2), feature ablation and other accessory functions.

Chapter 4

Results

In this chapter we describe the result of the degree project. First, we give a general overview of what has been built, then we also present some possible applications of Maggy, together with a comparison of similar frameworks.

4.1 Outcome

The result of this thesis work consists in the extension of Maggy: a framework for ablation studies and hyperparameter search, running on top of Apache Spark. Maggy has been successfully extended to support ablation studies in PyTorch, deep learning framework that is gaining a lot of popularity in the ML community in the last recent years, for the reasons illustrated in Chapter 1. Feature ablation has been generalized to all type of data rather than simple dataset columns, and the logic for ablation has been simplified as explained in Section 3.6. Finally, an API alternative to the one currently implemented in Maggy has been discussed, together with its pros and cons. This work also sheds some light on the virtues and limitations of PyTorch, especially in the context of building a framework on top of it, exploring in particular the flexibility of this tool, which has been harnessed to enable a wider variety of application, compared to the previous version of Maggy, that we will discuss in Section 4.2.

4.2 Applications

We are now going to illustrate a few relevant examples of what can be done with Maggy ablation. This list is not exhaustive and, in principle, all ablations that involve the modification of any data type (or layers of Sequential PyTorch models) have been enabled in Maggy.

4.2.1 Layer Ablation

In Section 3.4.2 we showed how to wrap a PyTorch sequential model in a function that will be used by Maggy to generate all the other ablated models that the user specifies. We will now compare that same initial model that we defined in Listing 3.9 with the ones produced after some of the ablation trials illustrated in Listing 3.10. In Listing 4.1 we can see the string representation our initial Sequential model, while in Listing 4.2 we reported the same model after two layer ablations: the first one includes layers 2 and 3, and the second one includes layers 4 and 5. Since all the trials are executed in parallel, they all remove components from the initial model and they do not harness the results of previously finished trials. The first ablation just removes the layer and its subsequent activation function without modifying other layers in the network, while the second one applies a correction to the Linear layer 6 (which after ablation becomes layer 4) to match its *input_features* to the *output_features* of layer 2 as it has been explained in Section 3.2.2.

```
>>> Original model:
 Sequential(
  (0): Linear(in_features=6, out_features=64, bias=True)
  (1): ReLU()
  (2): Linear(in_features=64, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=32, bias=True)
  (5): ReLU()
  (6): Linear(in_features=32, out_features=2, bias=True)
  (7): ReLU()
  (8): Linear(in_features=2, out_features=1, bias=True)
  (9): Sigmoid()
)
```

Listing 4.1: Initial PyTorch model

```
>>> Ablating layer 3 - ReLU()
>>> Ablating layer 2 - Linear(in_features=64, out_features=64, bias=True)
Sequential(
  (0): Linear(in_features=6, out_features=64, bias=True)
  (1): ReLU()
  (2): Linear(in_features=64, out_features=32, bias=True)
  (3): ReLU()
  (4): Linear(in_features=32, out_features=2, bias=True)
  (5): ReLU()
  (6): Linear(in_features=2, out_features=1, bias=True)
  (7): Sigmoid()
)

>>> Ablating layer 5 - ReLU()
>>> Ablating layer 4 - Linear(in_features=64, out_features=32, bias=True)
Sequential(
  (0): Linear(in_features=6, out_features=64, bias=True)
  (1): ReLU()
  (2): Linear(in_features=64, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=2, bias=True)
  (5): ReLU()
  (6): Linear(in_features=2, out_features=1, bias=True)
  (7): Sigmoid()
)
```

Listing 4.2: Resulting models after two different layer ablation

4.2.2 Column Ablation

A first simple scenario is the one that has already been presented in Chapter 3, that is the ablation of columns of tabular datasets. This is also a very common scenario as tabular datasets are quite frequent in ML. In Maggy, the user just needs to pass a Pandas DataFrame to the ablation study and specify which columns are to be included in it, as we illustrated in Listings 3.8 and 3.10. After feature ablation in a tabular dataset we expect to see the same dataset minus a column that we excluded: in the case of Titanic dataset with the ablation of feature "Pclass" here is the outcome that we obtain (some columns were omitted to fit the representation of this dataset to the page).

Id	Survived	Pclass	Name	Sex	Age	Fare
1	0	3	“Braund & Mr. Owen Harris”	m	22	7.25
2	1	1	“Cumings, Mrs. John Bradley”	f	38	71.2833
3	1	3	“Heikkinen, Miss. Laina”	f	26	7.925
...	...	...	...	...	...	...

Table 4.2.1: Sample from Titanic dataset

Id	Survived	Name	Sex	Age	Fare
1	0	“Braund & Mr. Owen Harris”	m	22	7.25
2	1	“Cumings, Mrs. John Bradley”	f	38	71.2833
3	1	“Heikkinen, Miss. Laina”	f	26	7.925
...	...	...	...	...	...

Table 4.2.2: Same sample as Table 4.2.1 after ablation of feature “Pclass”

4.2.3 Channel Ablation

What if we are not dealing with a tabular dataset? For instance, many tasks in deep learning regard image captioning, image classification, audio or video analysis and they are generally solved through the use of CNNs, which are neural networks that use convolutional filters to learn pattern from the dataset. A convolutional filter is a tensor of arbitrary shape that contains some information learned from the training data: in the case of an image dataset, these features might be textures, colors, shapes or even clear pictures (like animals, humans, cars, according to the entity of our dataset).

If we have an RGB[1] set of images to train our model we might want to test if all channels of our dataset are useful for the training or if we can lighten the model by excluding of some of them. In this case the user just needs to define the input shape of the training data after the ablation, together with the function that actually performs the ablation, reported in Listing 4.3. In Listing 4.4 we can see the outcome of feature ablation of the first channel on the CIFAR10[2] dataset. Since the images in the dataset are of size $32 \times 32 \times 3$, we just printed a portion of the first image for the sake of space on the page, but a visual representation of this process can be observed also in Figure 4.2.1.

```python
def ablate_feature(self, feature):
    if feature == "ch1":
        self.data = np.delete(self.data, obj=0, axis=3)
```

[1]an RGB image is made of three layers, each one representing levels of red, blue and green in the picture

[2]https://www.cs.toronto.edu/ kriz/cifar.html

```
    if feature == "ch2":
        self.data = np.delete(self.data, obj=1, axis=3)
    if feature == "ch3":
        self.data = np.delete(self.data, obj=2, axis=3)
```

Listing 4.3: Example of channel ablation

```
>>> Ablating features: ch1
# Before feature ablation
 [[[ 59  62  63]
  [ 43  46  45]
  [ 50  48  43]
  ...
  [158 132 108]
  [152 125 102]
  [148 124 103]]

 ...


# After feature ablation
 [[[ 62  63]
  [ 46  45]
  [ 48  43]
  ...
  [132 108]
  [125 102]
  [124 103]]

 ...
```

Listing 4.4: First image of CIFAR10, not yet normalized, before and after feature ablation.

4.2.4 Word Masking

Even though we always refer to ablation as the deletion of something, we can generalize this concept to the exclusion of certain features of our model/dataset even when we modify them instead of cancelling them. Here is an example that will clarify this section: imagine that you want to train a transformer [41] to recognize positive and negative sentiment in English sentences (this task is commonly known as "Sentiment

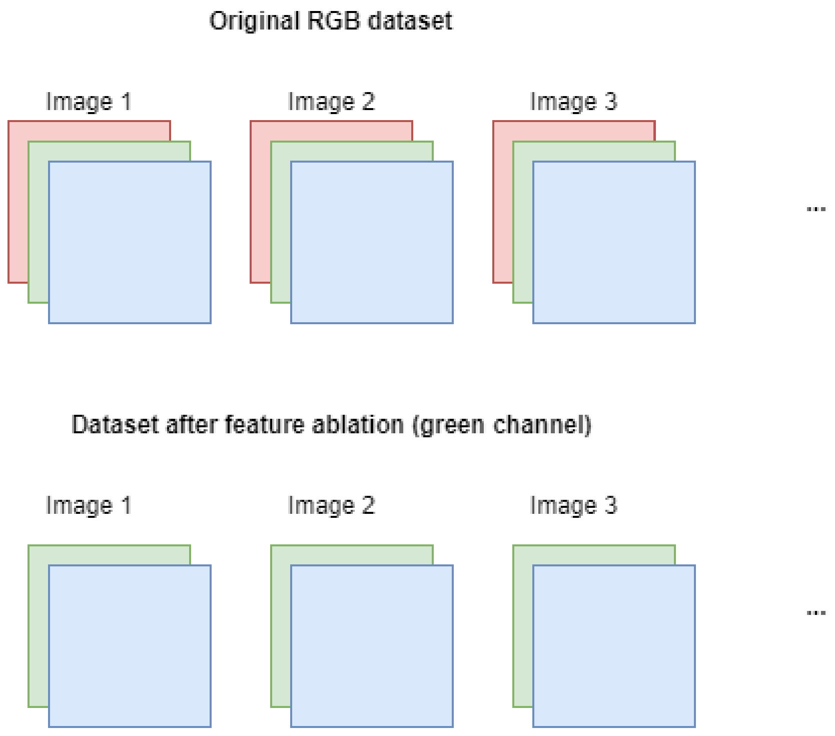

Figure 4.2.1: Visual represenation of channel ablation in an RGB dataset

Analysis” and it is quite popular in NLP). Chances are that you already have a pre-trained model that has a grasp of the English language that will map the words of a sentence to feature vectors (also known as “embeddings”) that in turn will be used to train a classifier that can split these vectors into two classes: positive and negative. Without getting too bogged down in the details, the intuition here is that the words that we used to train the classifier are the discriminant element for a sentence to recognized as positive or negative.

For instance, if we are dealing with articles taken from financial newspapers and in our dataset a certain bank ABC was going through a period of crisis, many articles in training dataset will report Bank ABC associated with a negative sentiment and the model is likely to learn that association (“Bank ABC is present in the article” $\Longrightarrow$ ”Negative sentiment”). Now, after one year, Bank ABC is doing well and the crisis is only a distant memory, but our model is up and running with training data sampled during the crisis, so the prediction might still be biased (and therefore erroneous). What can we do to prevent this bias? A simple trick is to “mask” bank names in the article by replacing them with a placeholder (e.g. the word “bank”): in this way we are making explicit to the model that it should not focus on the bank name to make a prediction.

Here in Listing 4.5 is reported a snippet of code that shows how to do this.

```python
def ablate_feature(self, feature):
    banks = ["Bank A", "Bank B", "Bank C"] # List of bank names to mask

    if feature == "bank_names":
        for article in self.data:
            for bank in banks:
                article.replace(bank, "placeholder")
```

Listing 4.5: Example of word masking

Of course, this does not prevent the user to define at the same time other types of ablation in the same function. The only thing that is necessary is another if branch in the code. In Tables 4.2.3 and 4.2.4 we can see an extract of the dataset Financial Phrase Bank [24] before and after applying word masking.

Sentiment	Sentence
neutral	“According to Gran , the company has no plans to move all production to Russia , although that is where the company is growing .”
negative	“The international electronic industry company Elcoteq has laid off tens of employees from its Tallinn facility.
positive	With the new production plant the company would increase its capacity to meet the expected increase in demand and would improve the use of raw materials and therefore increase the production profitability .

Table 4.2.3: Sample from Financial Phrase Bank

4.3 Comparison with Other Existing Frameworks

As we already mentioned, to the best of our knowledge there are no specific frameworks for ablation studies other than Maggy. Although, there are many available frameworks for NAS and HO. Since we are not focusing on HO in this thesis, we will now make a comparison with some of the most popular frameworks for NAS, which can coincide with an ablation study if the latter involves only the model architecture (and not the features).

Sentiment	Sentence
neutral	“According to placeholder , the company has no plans to move all production to Russia , although that is where the company is growing .”
negative	“The international electronic industry company placeholder has laid off tens of employees from its Tallinn facility.
positive	With the new production plant the company would increase its capacity to meet the expected increase in demand and would improve the use of raw materials and therefore increase the production profitability .

Table 4.2.4: Sample from Financial Phrase Bank after word masking

4.3.1 Autokeras

Autokeras [15] is a library developed to find the best ML model given a specific task, and together with Keras-tuner [16] they are among the most popular libraries for AutoML. However, Keras Tuner is quite a mature tool, while for Autokeras the authors state explicitly that the project is only a pre-release and it is far from being a finished product.

4.3.2 ENAS

ENAS (Efficient Neural Architecture Search) [7, 8] is a library available for PyTorch and TensorFlow that is used to discover good configurations of parts of some particular neural networks. It is based on the homonymous paper [36] that presents the idea of a graph search in a *Directed Acyclic Graph (DAG)* representing different neural architectures harnessing parameter sharing to speed up the computation. The implementation in TensorFlow supports already RNN and CNN cells, while in the PyTorch implementation only RNN are supported at the moment.

4.3.3 NNI

NNI [30] (Neural Network Intelligence) is a framework developed by Microsoft for NAS, HO, feature engineering and model compression. It is quite a complete toolkit for AutoML but regarding architecture search it provides only some benchmark to select between a series of candidate models or candidate operations to tweak the initial architecture.

4.3.4 Other Frameworks

As I write this thesis, probably a lot more frameworks are being developed for NAS and AutoML in general, because it is a hot topic that is gaining a lot of attention lately, especially because the ML community is realizing that relying on automated searches for neural architectures is often much faster and optimized than exploring them manually.

Chapter 5

Conclusions and future work

In this chapter we are going to finalize the dissertation drawing the conclusions: first a general summary is presented, then, we show a few considerations on the results obtained, and finally we list some possible prompts for future work.

5.1 Conclusion

In this thesis project we presented how to build a framework for ablation studies on top of PyTorch. We listed its key points which are efficiency, ease of use and minimal code changes (compared to normal PyTorch code), and we illustrated the main challenges and interesting points in the development. We showed some examples of how Maggy can be used for ablation studies, highlighting in particular its flexibility, as well as its simplicity for recurrent use cases (e.g. feature ablation of Pandas DataFrames). We mentioned the impact of PyTorch in the deep learning community as the motivation for the choice of the topic of this thesis, and we presented the comparison of Maggy with other popular tools for NAS.

5.2 Considerations on the Results

Looking back at the initial research question presented in Chapter 1 ("Can we provide an intuitive API for PyTorch users to efficiently conduct ablation studies with minimal code changes compared to normal PyTorch code?") we should now try to answer it. Intuitiveness and efficiency are hard to quantify: we can all agree that it is easy to recognize a really intuitive interface but it is quite difficult to agree on what really makes something easy to use. On the other hand, even though efficiency seems like an easily quantifiable property, it is not (especially in deep learning research) because of the lack of common benchmarks, as highlighted by Lindauer & Hutter [19]: every new paper affirms to be the state of the art but often there is very little or no comparison to other similar work and, even when there is, there are so many parameters to take into account (such as hardware, datasets, hyperparameters, etc.) that tracing parallels is almost impossible for researchers. When other computer scientists try to replicate the results obtained in previous literature, matching the setups of these studies is quite a hard task, and consequently the replicated results differ a lot. In any science, and therefore also computer science, replicability is a key point for any good work, and if we can't ensure that we can not ensure high quality research in AI.

So, what can really define the quality of a framework? Maybe adoption, impact and in general the engagement of the community is what defines success best in this case. Unfortunately, the extended version of Maggy described in this thesis has not been made publicly available yet, so I can not include any data on the framework adoption.

5.3 Future Work

Here are listed some ideas for some possible future work:

- Include support for functional models: what has been developed so far leverages some principles that belong only to sequential models, especially to match the feature coherence between layers after the ablation. It would be interesting to add support for PyTorch functional models since they are also widely adopted in the ML community.
- Speed up trials with even more parallelization: PyTorch is optimized for parallelization through CUDA [32]. Even though Maggy already executes trials in parallel through Spark, one trial is assigned to one executor. Since it might happen that certain trials are particularly heavy computationally speaking, we can explore what happens if we assign a single trial to multiple executors optimizing the workload with CUDA, or even performing distributed training on top of Apache Spark.

- Leverage early-stopping for ablation studies: at the moment, Maggy supports early-stopping only for hyperparameter search through median rule. It would be interesting to explore the amount of resources that can be saved by defining an early-stopping strategy for the ablation studies that do not perform well enough, in favour of the more promising ones.
- Universal ablation: this thesis work enable Maggy to ablate any type of data, in any format, provided that the user specifies how that ablation will happen. Still, we can take another step further and re-think ablation for literally anything. To clarify this, the paper on DistilBERT [11] (a popular model in NLP) has a section where it describes the ablation of single parts of a triplet loss, while the paper "The unreasonable effectiveness of the forget gate" [43] describes indeed the ablation of a forget gate in an *Long-Short Term Memory (LSTM)*.

Bibliography

[1] Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S., Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Goodfellow, Ian, Harp, Andrew, Irving, Geoffrey, Isard, Michael, Jia, Yangqing, Jozefowicz, Rafal, Kaiser, Lukasz, Kudlur, Manjunath, Levenberg, Josh, Mane, Dan, Monga, Rajat, Moore, Sherry, Murray, Derek, Olah, Chris, Schuster, Mike, Shlens, Jonathon, Steiner, Benoit, Sutskever, Ilya, Talwar, Kunal, Tucker, Paul, Vanhoucke, Vincent, Vasudevan, Vijay, Viegas, Fernanda, Vinyals, Oriol, Warden, Pete, Wattenberg, Martin, Wicke, Martin, Yu, Yuan, and Zheng, Xiaoqiang. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: `1603.04467 [cs.DC]`.

[2] Akiba, Takuya, Sano, Shotaro, Yanase, Toshihiko, Ohta, Takeru, and Koyama, Masanori. “Optuna”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining* (July 2019). DOI: `10.1145/3292500.3330701`. URL: `http://dx.doi.org/10.1145/3292500.3330701`.

[3] Apache Software Foundation. *Hadoop*. URL: `https://hadoop.apache.org`.

[4] Bengio, Yoshua, Deleu, Tristan, Rahaman, Nasim, Ke, Rosemary, Lachapelle, Sébastien, Bilaniuk, Olexa, Goyal, Anirudh, and Pal, Christopher. “A Meta-Transfer Objective for Learning to Disentangle Causal Mechanisms”. In: (2019). arXiv: `1901.10912 [cs.LG]`.

[5] *Brief history of Neural Networks*. URL: `https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html`.

[6] Chollet, François et al. *Keras*. `https://keras.io`. 2015.

[7] *ENAS for PyTorch*. URL: `https://github.com/carpedm20/ENAS-pytorch`.

[8] *ENAS for TensorFlow*. URL: `https://github.com/MINGUKKANG/ENAS-Tensorflow`.

[9] Feurer, Matthias, Eggensperger, Katharina, Falkner, Stefan, Lindauer, Marius, and Hutter, Frank. *Auto-Sklearn 2.0: The Next Generation*. 2020. arXiv: `2007.04074 [cs.LG]`.

[10] Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[11] Hinton, Geoffrey, Vinyals, Oriol, and Dean, Jeff. *Distilling the Knowledge in a Neural Network*. 2015. arXiv: `1503.02531 [stat.ML]`.

[12] *How to stop data centres from gobbling up the world's electricity*. 2018. URL: `https://www.nature.com/articles/d41586-018-06610-y`.

[13] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, eds. *Automated Machine Learning: Methods, Systems, Challenges*. In press, available at http://automl.org/book. Springer, 2018.

[14] Ismail, Mahmoud, Gebremeskel, Ermias, Kakantousis, Theofilos, Berthou, Gautier, and Dowling, Jim. "Hopsworks: Improving User Experience and Development on Hadoop with Scalable, Strongly Consistent Metadata". In: June 2017, pp. 2525–2528. DOI: `10.1109/ICDCS.2017.41`.

[15] Jin, Haifeng, Song, Qingquan, and Hu, Xia. "Auto-Keras: An Efficient Neural Architecture Search System". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining* (July 2019). DOI: `10.1145/3292500.3330648`. URL: `http://dx.doi.org/10.1145/3292500.3330648`.

[16] *Keras Tuner GitHub page*. URL: `https://github.com/keras-team/keras-tuner`.

[17] Li, Lisha, Jamieson, Kevin, DeSalvo, Giulia, Rostamizadeh, Afshin, and Talwalkar, Ameet. *Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization*. 2016. arXiv: `1603.06560 [cs.LG]`.

[18] Lin, Tsung-Yi, Maire, Michael, Belongie, Serge, Bourdev, Lubomir, Girshick, Ross, Hays, James, Perona, Pietro, Ramanan, Deva, Zitnick, C. Lawrence, and Dollár, Piotr. *Microsoft COCO: Common Objects in Context*. 2014. arXiv: 1405.0312 [cs.CV].

[19] Lindauer, Marius and Hutter, Frank. "Best Practices for Scientific Research on Neural Architecture Search". In: (2019). arXiv: 1909.02453 [cs.LG].

[20] Liu, Hongyu and Lang, Bo. *Machine Learning and Deep Learning Methods for Intrusion Detection Systems: A Survey*. Oct. 2019. DOI: 10.3390/app9204396. URL: https://www.researchgate.net/publication/336630211/figure/fig2/AS:815172427984898@1571363660489/Taxonomy-of-machine-learning-algorithms.png.

[21] *Logical Clocks AB*. URL: https://www.logicalclocks.com/.

[22] *Maggy, Earlystop Module*. URL: https://github.com/logicalclocks/maggy/tree/master/maggy/earlystop.

[23] *Maggy, Logical Clocks Github repository*. URL: https://github.com/logicalclocks/maggy.

[24] Malo, Pekka, Sinha, Ankur, Takala, Pyry, Korhonen, Pekka, and Wallenius, Jyrki. "Good Debt or Bad Debt: Detecting Semantic Orientations in Economic Texts". In: *Journal of the American Society for Information Science and Technology* (Apr. 2014). DOI: 10.1002/asi.23062.

[25] *Marie-Jean-Pierre Flourens, Encyclopedia Britannica*. URL: https://www.britannica.com/biography/Marie-Jean-Pierre-Flourens.

[26] Medium. 2017. URL: https://cdn-images-1.medium.com/max/1600/1*RE6XwUu9YA6DwFKIJ8ImWQ.png.

[27] Medium. 2019. URL: https://miro.medium.com/max/1700/0*_SH7tsNDTkGXWtZb.png.

[28] Meister, Moritz Johannes. "Maggy: open-source asynchronous distributed hyperparameter optimization based on Apache Spark". In: (2019).

[29] Meister, Moritz, Sheikholeslami, Sina, Andersson, Robin, Ormenisan, Alexandru A, and Dowling, Jim. “Towards Distribution Transparency for Supervised ML With Oblivious Training Functions”. In: *Workshop on MLOps Systems, International Conference on Machine Learning and Systems (MLSys)*. 2020.

[30] *Microsoft NNI*. URL: `https://github.com/Microsoft/nni`.

[31] Nature. 2018. URL: `https://media.nature.com/lw800/magazine-assets/d41586-018-06610-y/d41586-018-06610-y_16109962.png`.

[32] Nickolls, John, Buck, Ian, Garland, Michael, and Skadron, Kevin. “Scalable Parallel Programming with CUDA”. In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: `10.1145/1365490.1365500`. URL: `https://doi.org/10.1145/1365490.1365500`.

[33] O., Russakovsky, J., Deng, and al., Su H. et. *ImageNet Large Scale Visual Recognition Challenge, Int J Comput Vis 115, 211–252*. 2015. URL: `https://doi.org/10.1007/s11263-015-0816-y`.

[34] *Official Apache Spark documentation*. URL: `https://spark.apache.org/`.

[35] Paszke, Adam, Gross, Sam, Massa, Francisco, Lerer, Adam, Bradbury, James, Chanan, Gregory, Killeen, Trevor, Lin, Zeming, Gimelshein, Natalia, Antiga, Luca, Desmaison, Alban, Köpf, Andreas, Yang, Edward, DeVito, Zach, Raison, Martin, Tejani, Alykhan, Chilamkurthy, Sasank, Steiner, Benoit, Fang, Lu, Bai, Junjie, and Chintala, Soumith. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: `1912.01703 [cs.LG]`.

[36] Pham, Hieu, Guan, Melody Y., Zoph, Barret, Le, Quoc V., and Dean, Jeff. *Efficient Neural Architecture Search via Parameter Sharing*. 2018. arXiv: `1802.03268 [cs.LG]`.

[37] *ScaledML Conference 2020*. URL: `https://www.youtube.com/watch?v=HBqCpWldPII&list=PLRM2gQVaW_wVM4FBALoM7Wydb6n9NOiJO&index=9&t=0s&ab_channel=Matroid`.

[38] Schwartz, Roy, Dodge, Jesse, Smith, Noah A., and Etzioni, Oren. “Green AI”. In: (2019). arXiv: `1907.10597 [cs.CY]`.

[39] TURING, A. M. "I.—COMPUTING MACHINERY AND INTELLIGENCE". In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf. URL: https://doi.org/10.1093/mind/LIX.236.433.

[40] van der Walt, S., Colbert, S. C., and Varoquaux, G. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science Engineering* 13.2 (2011), pp. 22–30.

[41] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N., Kaiser, Lukasz, and Polosukhin, Illia. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[42] Wang, Alex, Singh, Amanpreet, Michael, Julian, Hill, Felix, Levy, Omer, and Bowman, Samuel R. *GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding*. 2018. arXiv: 1804.07461 [cs.CL].

[43] Westhuizen, Jos van der and Lasenby, Joan. *The unreasonable effectiveness of the forget gate*. 2018. arXiv: 1804.04849 [cs.NE].