Ablation Programming for Machine Learning

SINA SHEIKHOLESALAMI
Ablation Programming for Machine Learning

SINA SHEIKHOLESAMI

Master in Computer Science
Date: August 6, 2019
Supervisor: Jim Dowling
Examiner: Seif Haridi
School of Electrical Engineering and Computer Science
Host company: RISE SICS AB & Logical Clocks AB
Swedish title: Ablation Programming för Maskininlärning
Abstract

As machine learning systems are being used in an increasing number of applications - from analysis of satellite sensory data and health-care analytics to smart virtual assistants and self-driving cars - they are also becoming more and more complex. This means that more time and computing resources are needed in order to train the models and the number of design choices and hyperparameters will increase as well. Due to this complexity, it is usually hard to explain the effect of each design choice or component of the machine learning system on its performance.

A simple approach for addressing this problem is to perform an ablation study, a scientific examination of a machine learning system in order to gain insight on the effects of its building blocks on its overall performance. However, ablation studies are currently not part of the standard machine learning practice. One of the key reasons for this is the fact that currently, performing an ablation study requires major modifications in the code as well as extra compute and time resources.

On the other hand, experimentation with a machine learning system is an iterative process that consists of several trials. A popular approach for execution is to run these trials in parallel, on an Apache Spark cluster. Since Apache Spark follows the Bulk Synchronous Parallel model, parallel execution of trials includes several stages, between which there will be barriers. This means that in order to execute a new set of trials, all trials from the previous stage must be finished. As a result, we usually end up wasting a lot of time and computing resources on unpromising trials that could have been stopped soon after their start.

We have attempted to address these challenges by introducing MAGGY, an open-source framework for asynchronous and parallel hyperparameter optimization and ablation studies with Apache Spark and TensorFlow. This framework allows for better resource utilization as well as ablation studies and hyperparameter optimization in a unified and extendable API.

Keywords: Distributed Machine Learning, Distributed Systems, Ablation Studies, Apache Spark, Keras, Hopsworks
**Sammanfattning**

Eftersom maskininlärningssystem används i ett ökande antal applikationer - från analys av data från satellitsensorer samt sjukvården till smarta virtuella assistenter och självkörande bilar - blir de också mer och mer komplexa. Detta innebär att mer tid och beräkningsresurser behövs för att träna modellerna och antalet designval och hyperparametrar kommer också att öka. På grund av denna komplexitet är det ofta svårt att förstå vilken effekt varje komponent samt designval i ett maskininlärningssystem har på slutresultatet.

En enkel metod för att få insikt om vilken påverkan olika komponenter i ett maskininlärningssystem har på systemets prestanda är att utföra en ablationsstudie. En ablationsstudie är en vetenskaplig undersökning av maskininlärningssystem för att få insikt om effekterna av var och en av dess byggstenar på dess totala prestanda. Men i praktiken så är ablationsstudier ännu inte vanligt förekommande inom maskininlärning. Ett av de viktigaste skälen till detta är det faktum att för närvarande så krävs både stora ändringar av kod för att utföra en ablationsstudie, samt extra beräknings- och tidsresurser.

Vi har försökt att ta itu med dessa utmaningar genom att använda en kombination av distribuerad asynkron beräkning och maskininlärning. Vi introducerar maggy, ett ramverk med öppen källkodsram för asynkron och parallell hyperparameteroptimering och ablationsstudier med PySpark och TensorFlow. Detta ramverk möjliggör bättre resursutnyttjande samt ablationsstudier och hyperparameteroptimering i ett enhetligt och utbyggbart API.
# Contents

1 Introduction ........................................... 1
  1.1 Motivation ....................................... 1
  1.2 Problem Statement ............................... 2
  1.3 Goals and Requirements ......................... 3
  1.4 Methodology ..................................... 4
  1.5 Ethics and Sustainability Aspects ............ 4
  1.6 Thesis Contributions ........................... 4
  1.7 Outline of the Thesis .......................... 5

2 Background ............................................ 6
  2.1 Machine Learning and Deep Learning ......... 7
    2.1.1 Data, Model, and Learning ............... 7
    2.1.2 Artificial Neural Networks ............... 11
    2.1.3 Convolutional Neural Networks .......... 13
  2.2 Ablation Study .................................. 15
  2.3 Platforms and Frameworks ....................... 17
    2.3.1 Apache Spark ................................ 17
    2.3.2 Keras and TensorFlow ....................... 18
    2.3.3 Hopworks .................................. 19
  2.4 Back to the Main Problem ...................... 20

3 Design and Implementation ......................... 21
  3.1 Requirements, Assumptions, and Goals ....... 21
    3.1.1 Framework General Requirements ........ 22
    3.1.2 Specific Requirements for the Ablation API .. 24
    3.1.3 Summary of Requirements ................ 24
  3.2 MAGGY Core .................................... 25
  3.3 Ablation on MAGGY ............................. 28
    3.3.1 Ablation Policies .......................... 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2</td>
<td>Implementing MAGGY Ablation</td>
<td>30</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Implementing the LOCO Policy</td>
<td>31</td>
</tr>
<tr>
<td>3.3.4</td>
<td>User API</td>
<td>32</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Developer API</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>Results and Discussion</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Addressing the Goals and Requirements</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison with Other Frameworks</td>
<td>39</td>
</tr>
<tr>
<td>5</td>
<td>Conclusion and Future Work</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Conclusion</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>Future work</td>
<td>42</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Core MAGGY Platform</td>
<td>42</td>
</tr>
<tr>
<td>5.2.2</td>
<td>MAGGY Ablation</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>46</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Simplified end-to-end machine learning process</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>MNIST data example</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Hyperparameter tuning is an iterative process</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Simple multilayer perceptron</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>LeNet-5 architecture</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Components of a machine learning system</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Hopsworks unified analytics platform</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Examples of ablation trials</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Un-directed vs. Directed experiments on Spark</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>High-level view of the architecture of MAGGY</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>MAGGY Ablation User API workflow</td>
<td>33</td>
</tr>
</tbody>
</table>
List of Source Codes

1. Creating an AblationStudy instance. . . . . . . . . . . . . . . 33
2. Adding components to the ablation study. . . . . . . . . . . 34
3. Defining the base model generator. . . . . . . . . . . . . . . 34
4. Wrapping the user code in a function. . . . . . . . . . . . . . 35
5. Launching the experiment. . . . . . . . . . . . . . . . . . . . 36
6. Abstract methods for implementing custom ablators. . . . . 36
List of Tables

4.1 Comparing MAGGY with keras-tuner and LOFO Importance 40
Chapter 1

Introduction

This thesis discusses the design and implementation of MAGGY, an open-source asynchronous computing framework on top of Apache Spark, as well as an intuitive Application Programming Interface (API) that enables machine learning researchers and practitioners to perform ablation studies for machine learning. In this Chapter, we will briefly discuss the motivation and the context of the problem, research goals and methodology, the ethical and sustainability aspects of the work, and the contributions and the outline of the thesis.

1.1 Motivation

Machine Learning and deep learning are the cornerstones of many recent breakthroughs and innovations. Their use cases range from on-spot translations between languages, to self-driving cars and accurate disease detection from medical imagery. This extraordinary success has led to a lot of research activities that aim to develop better, faster, and more accurate algorithms and models, for more applications. As a consequence, research groups and practitioners all around the world engage in an iterative process of designing, implementing, and tuning the performance of new machine learning and deep learning algorithms on a daily basis. This iterative process involves launching several trials, where each trial is different in terms of the configuration of the model, the hyperparameters of the model, or the dataset used for training, among other things.

Hyperparameter Tuning (also referred to as hyperparameter optimization) is a well-known example of such experiments. A hyperparam-
eter of the model, is a parameter that its value is not learned through the training process, but choosing the right value for it is crucial in order to get a good performance. There are several approaches for hyperparameter optimization, many of which fall under the broader category of black-box optimization. There are also simpler approaches like random search and grid search. However, in all these approaches, we run many trials, each with a specific configuration for hyperparameters, and then observe the resulting performance metrics to decide on good hyperparameter values.

Another example of experimentation with our machine learning systems follows the fact that new architectures of neural networks are usually a result of adding new layers or new types of connections between the neurons or layers. Hence, based on a research practice that is well-known in physiological and medical research, high-quality literature in the fields of neural networks and machine learning sometimes include a section on ablation studies. In this context, we define ablation study as a scientific examination of a machine learning system in order to gain insight on the effects of each of its building blocks on its overall performance. This examination involves removal of the components of the system, be it specific layers, neurons, or connection links, or even the features of the dataset that the model is being trained on. However, though it seems to be a simple and intuitive practice, performing an ablation study is still not part of the standard research practice in machine learning and deep learning.

### 1.2 Problem Statement

Experimentation with a machine learning system is an iterative process that consists of several trials. A popular approach for execution is to run these trials in parallel, on an Apache Spark [2] cluster. However, the Bulk Synchronous Parallel execution model of Apache Spark means that running parallel Machine Learning and Deep Learning trials on a Spark cluster consists of stages that require all executors to finish their tasks before a new stage could be started, therefore introducing barriers between stages. In the context of experimentation with machine learning and deep learning systems, these barriers result in low resource utilization. For example, in a hyperparameter optimization experiment, we can soon find out whether a given configuration
is good enough to continue evaluating or should be stopped since it is not promising. Similarly, in an ablation study experiment, execution of some trials may require much less time than some other trials since e.g. the model that has to be trained may be less complex. However, because of the stage-based, synchronous execution model of Apache Spark, all trials in a stage should be finished before a new set of trials can be started.

Thus, a framework on top of Apache Spark that enables early-stopping and global control of trials, all without modifying Spark itself, would be much desirable as it allows for higher resource utilization as well as the possibility of exploiting the global status of the trials at any given point during an experiment to modify existing trials or generate new ones.

Given such a framework, it also desirable to develop a simple, extendable Application Programming Interface (API) for performing ablation study experiments. In order to be picked up by the community, the API should follow and exploit best-practices in developing machine learning code, and it should not require much modification to the existing code that is used for training or evaluating a model. An API with the aforementioned specifications can help for inclusion of ablation studies in the standard machine learning practice.

### 1.3 Goals and Requirements

In this work, we attempt to address the stated problem by fulfilling the following primary requirements and goals:

1. Improving resource utilization of parallel execution of machine learning and deep learning trials on Apache Spark clusters by adding support for global early-stopping and global control of the experiment, therefore eliminating the wasted compute between the moment a trial is finished and the time when a new stage is started.

2. Designing and Implementing a user-friendly and extensible API for performing ablation studies of machine learning systems.

These primary requirements, as well as other important goals of this thesis are discussed in more detail in Section 3.1.
1.4 Methodology

The work for this thesis has been carried out following the qualitative research methodology, since our primary objective is to design and develop a framework and an API that allows for easy experimentation with machine learning systems while increasing the resource utilization of the underlying computing environment. The full project included two main deliverables, each corresponding to one of the primary goals and requirements mentioned in Section 1.3. Throughout the project, study of recent literature was combined with design and development of prototypes and proof-of-concepts.

1.5 Ethics and Sustainability Aspects

The thesis addresses problems that have direct connection to ethical, environmental, and societal issues we face as machine learning is being used more and more in our everyday lives. On one hand, higher resource utilization means less power and other resources will be used for performing machine learning experiments and research. On the other hand, inclusion of ablation studies in the standard machine learning practice is a simple yet important step towards a better understanding of how our machine learning systems work.

Also, better understanding of the effects of different features of datasets on the performance of a machine learning system may lead to elimination of features of the dataset that are not so crucial for the system. Hence, we can be more restrictive in collecting or using sensitive information of individuals or entities, and only use what is really needed.

1.6 Thesis Contributions

The contributions of the thesis include the design and development of Maggy [1], an open-source framework for asynchronous computation on top of Apache Spark, as well as an API on top of Maggy for performing ablation studies of machine learning systems. Maggy is released under the AGPL-3.0 license [3] and hosted on GitHub. Maggy satisfies the requirements discussed in Section 1.3 and we believe by unifying the API for ablation studies with the API for hyperparameter
optimization, we have taken a major step towards the inclusion of ablation studies in the standard machine learning research practice.

1.7 Outline of the Thesis

This thesis is organized as follows. In Chapter 2, we provide the background necessary for understanding the problem and the contributions of this thesis. Specifically, in Section 2.2, we present a definition of ablation studies in the context of machine learning. We also briefly introduce the platforms and frameworks that this work is built upon. Chapter 3 is dedicated to a detailed discussion on the design and implementation of MAGGY and its Ablation API. Then, in Chapter 4, we briefly explain how our solution satisfies the goals of this project, and compare MAGGY with two similar, well-known frameworks. We then conclude this thesis with Chapter 5, where we provide a final discussion of the impact of our work, its limitations, and the directions for further research.
Chapter 2

Background

Hyperparameter tuning and ablation studies are two particular tasks in the practice of machine learning and deep learning. This thesis introduces new approaches for performing these tasks using a framework built on top of Apache Spark [2], TensorFlow [4], and the Hopsworks Big Data and AI platform [5]. Therefore, this Chapter aims to provide the reader with the necessary background for following and understanding the contributions of the thesis.

First, a short introduction to machine learning and deep learning will be presented. In particular, we will discuss Convolutional Neural Networks (CNNs), as the state-of-the-art architecture for many deep learning applications such as computer vision and image recognition.

After that, we will discuss the platforms and systems that our framework is based upon. Maggy and its corresponding APIs are built on top of Apache Spark and Hopsworks, and support models and codes developed for TensorFlow’s implementation of the Keras API [6]. Hence, we will briefly introduce these platforms, their limitations, and their potentials that we have exploited.

Finally, a brief introduction to Black-box optimization and Ablation studies will be presented, as this thesis contributes to these two stages in machine learning workflows.

Readers with background in machine learning and deep learning can skip the first part of this Chapter and continue from Section 2.2 where we define and set the context of ablation studies in machine learning for the rest of this thesis.
CHAPTER 2. BACKGROUND

2.1 Machine Learning and Deep Learning

Machine Learning has been around for decades now, and the term has been used at least since 1959 when Arthur Samuel developed a program that was able to "learn" to play checkers [7]. Hence, it is not surprising that over time different definitions and categorizations of machine learning systems and tasks have been proposed. In the introduction to his quintessential book, *Machine Learning* [8], Tom M. Mitchell suggests the following formal definition:

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

Goodfellow et al., in their *Deep Learning* book [9], refer to machine learning as the capability of Artificial Intelligence (AI) systems "to acquire their own knowledge, by extracting patterns from raw data." Computer programs and AI systems then leverage this knowledge to perform a variety of tasks, such as recognizing handwritten digits, predicting house prices based on different properties - or features - of a number of previously sold houses, detecting fraudulent financial transactions in real-time, recommending products to customers of an e-commerce website, predicting future climate metrics based on past and current remote sensing data and satellite images [10], and translating texts between different languages.

It is worth noting that right now the majority of the machine learning and AI systems are task-specific, meaning they are only designed for solving single, specific tasks (or a small set of closely related tasks). However, these task-specific systems have already outperformed humans in a variety of problems, and can be considered as the building blocks to Artificial General Intelligence, which for long has been a defining goal of AI research [11, 12].

2.1.1 Data, Model, and Learning

Although each machine learning system is designed for a specific task and these tasks form a very broad spectrum (see above for some examples), it is possible to describe these systems in terms of data, models, and learning [13]. Simply put, the goal of a machine learning system
is to learn from historical data (or interactions in a controlled environment) for optimizing a model in order to perform well on unseen data (or an unknown environment).

To expand on that, one way to formulate a learning problem is by defining it as a mapping from available or possible inputs to desired outputs (or target variables). The main input of our machine learning system is a collection of historical or real-time data that can be structured or unstructured, and is referred to as the dataset. We then select a suitable machine learning model. The model can be thought of as a black-box that contains a number of parameters, and can map instances from the input space to the output space. An example of a machine learning model is Simple Linear Regression which can be used for example for predicting house prices based on previous sales data and some attributes of a house (e.g. its age and number of bedrooms).

Next, we train our machine learning system: using a machine learning algorithm, we optimize the parameters of our model with respect to a loss function that we use to evaluate the performance of the model, for example, the difference between the predicted outputs and the actual correct outputs for the input data. We call this difference the training error. We usually feed the input data to the model in smaller subsets known as batches, and we call one complete pass over the data an epoch. We repeat this process until we achieve a desired performance. One way to specify the desired performance is to select one or more performance metrics (e.g. accuracy) and their corresponding thresholds.

![Figure 2.1: A simplified view of an end-to-end machine learning process.](image)

This iterative process is known as the learning or training process and is depicted in Figure 2.1. Note that in this figure we show the dataset to be in form of a database, but this is only one of the various possible formats that the data might come in.
However, much of this process can be automated, and in fact this is the focus of the field of Automated Machine Learning or AutoML [14].

Sometimes the instances of this dataset are labeled, meaning that we know the corresponding correct outputs (or labels) for individual input instances. Machine Learning algorithms and systems that deal with this type of data are categorized under Supervised Learning. If we do not have a labeled dataset, and for example want to put data points into groups based on their similarities (a task known as clustering) or discover hidden structures within the dataset, the task may fall under Unsupervised Learning.

In this framework there are other categories of machine learning tasks and systems, such as Semi-supervised Learning in which the dataset contains both labeled and unlabeled data, and Reinforcement Learning where the machine learning system is called an agent who interacts with the environment (which can contain other agents) and gradually learns by receiving rewards or penalties as the result of its actions [15]. In this case we can think that the environment and our definitions of rewards and penalties provide the data.

An often-cited example of a machine learning task is recognizing handwritten digits, which is a subset of image processing and computer vision and falls under supervised learning. In this task, we have a dataset that consists of images of handwritten digits (see Figure 2.2) and would like to have a program that is able to automatically classify those images, or map hand writings to corresponding correct classes of digits from 0 to 9. To be able to assess (or evaluate) the performance of our program, we also have to decide upon one or more performance metrics. For example, we might use the number of correctly recognized digits averaged by the total number of image instances. This is a simple measure known as accuracy, and is commonly used for evaluating machine learning models.
Chapter 2. Background

Partitioning the Dataset into Training, Test, and Validation Sets

When we deal with a labeled dataset\(^\text{1}\), we usually partition it into a training set, a test set, and (often) a validation set.

Training set is used as the input data for learning the values of (optimizing) the parameters of the machine learning model. However, it might be the case that the model performs very well on the training data but produces unsatisfactory results if given previously unseen data (remember that the goal of a machine learning system is to perform well on unseen data). This usually indicates that the model does not generalize well, and we say it has overfitted the training dataset.

Trivially, one way to find out if the model is overfitting or not is to place it out in the real world and observe its results, but this if often very costly - you do not want to put a self-driving car on the road only to find out it does not work well since it caused a serious traffic accident.

A much simpler approach is to take out a portion of the dataset and use it only for evaluating the performance of the model. By this we mean that the model does not see this part of the dataset during training. We call this partition the test set. Once we have trained the model for a number of iterations and think that we have a reasonable model, we evaluate its performance using the test set (by calculating the test error or generalization error with the loss function) and the performance metric(s).

In addition to the parameters of the model, which values will be optimized (learned) using the training data and the loss function, a machine learning system also has other parameters that are not learned during training. Examples of these parameters include the number of data instances in each batch (batch size), as well as learning rate, which specifies how much we should change the model based on an observation during training. These parameters are called hyperparameters and are usually tuned (optimized) manually by machine learning practitioners.

Tuning the hyperparameters of the model is also an iterative process and typically includes coming up with several combinations of values, training the models using each of these combinations, observing the results, and maybe using the insights for coming up with better combinations. Figure 2.3 depicts this iterative process. Again, to prevent

\(^{1}\)For unlabeled datasets or unsupervised learning tasks, depending on the problem, it may also be possible to partition the dataset into training, test, and validation sets.
overfitting, we may take out a third subset of the main dataset and use it to tune the hyperparameters. We call this subset the validation set.

2.1.2 Artificial Neural Networks

Although classic machine learning algorithms have achieved good results in many tasks, it is a known fact that their performance eventually saturates and does not increase as the amount of their input data is increased. However, a specific group of machine learning models and algorithms are known to get better and better as the amount of data they receive is increased. These models fall under the umbrella of Artificial Neural Networks. A specific subcategory of these models, known as Deep Neural Networks, constitutes the backbone of some of the most important breakthroughs in machine learning research in the previous decade.

Artificial Neural Networks (ANNs) are a family of machine learning models that are inspired by neural circuits in the brains of animals.

---

2 Sometimes, instead of putting aside a separate partition as the validation set, we use a technique called cross-validation. Cross-validation consists of multiple rounds, and in each we partition the training dataset into complementary subsets (folds), use a number of them for validation and the rest for training.

3 To be more precise, ANNs are generally considered as a type of computing systems, but we focus on their applications in the domain of machine learning since they are best known for this.
They consist of units called artificial neurons that are connected to each other in various patterns that are known as network architectures (or architectures for short). Each unit is associated with an activation function that determines the output of the unit given its input.

![A simple multilayer perceptron consisting of a single hidden layer with five units.](image)

**Figure 2.4:** A simple multilayer perceptron consisting of a single hidden layer with five units.

A famous class of ANNs is the Multilayer Perceptron (MLP), which in its simplest form consists of three layers of neurons: an input layer, a hidden layer, and an output layer. The layers of an MLP are fully connected, meaning that each neuron in a given layer receives input from all the neurons in its previous layer, and has its output connected to all the neurons in the next layer.

A simple MLP architecture is shown in Figure 2.4. This MLP is an example of an ANN with a single hidden layer. An ANN that has more than one single hidden layer is called a Deep Neural Network (DNN).

MLPs, however, have a number of shortcomings, among which is their inability to leverage topological and temporal characteristics of the input data. For example, they disregard the fact that nearby pixels in an image are correlated, or the order of inputs in time-series data is crucial in prediction tasks. As a consequence, different architectures have been developed over the years to take into account specific characteristics and properties of different tasks and problems.
These architectures and models, although still task-specific, have dramatically increased the performance of machine learning systems within their respective problem domains; however, this improvement has come with a cost of increased complexity. Models based on Convolutional Neural Networks (CNNs) for image recognition and Recurrent Neural Networks (RNNs) for sequence-based tasks such as natural language processing are perhaps the most well-known of these architectures.

### 2.1.3 Convolutional Neural Networks

Convolutional Neural Networks are a class of ANNs that take inspiration from the visual cortex of the brain. Studies by David Hubel and Torsten Wiesel in the later half of 1950s revealed the importance of local receptive fields for the neurons in the visual cortex [19]. It turned out that each neuron can be stimulated only by a limited region in the visual field. They also showed that some neurons can only react to certain structures and patterns, for example horizontal lines, and some neurons react to combinations of lower-level patterns.

These findings shaped the directions of the research in fields of biological cybernetics and artificial intelligence in the decades that followed [4] and in 1998 LeCun et al. presented LeNet-5 [21], an ANN architecture for recognition of handwritten digits (of the MNIST dataset, see Figure 2.2). The main building blocks of LeNet-5 were convolutional layers and pooling layers, inspired by local receptive fields. Convolutional layers detect patterns and structures in the data (images), and pooling layers are for subsampling. Subsampling reduces the size of each input (e.g. height and width for an image), hence the number of model parameters and its computational load. The architecture of the LeNet-5 network is depicted in Figure 2.5.

The receptive field for a convolutional layer is translated into convolution kernels, also known as filters in general. A filter is a trainable set of weights, and when applied to the whole image or its input coming from its previous layer, produces a representation of the input known as a feature map. A feature map highlights areas and structures in the image that activate the filter the most [20].

---

Each convolutional layer, in turn, is a stack of several feature maps. For example, in Figure 2.5 the first convolutional layer (C1) consists of six feature maps. Each filter highlights a certain structure or pattern, so having several convolutional layers in the model means the network can learn to recognize different patterns and structures and use them for image classification, object detection, or other related tasks.

LeNet-5 would take days to train, but achieved better performance on the MNIST dataset than the more "classic" machine learning algorithms of its time. LeCun et al. also discussed the consequences of increasing the size of the training dataset, and the variations in the architecture (e.g., increasing the number of hidden layers): When plenty of data is available, many methods can attain respectable accuracy. The neural-net methods run much faster and require much less space than memory-based techniques. The neural nets’ advantage will become more striking as training databases continue to increase in size.

In the years that followed, the increase in computing power (in particular, the introduction of the general-purpose Graphical Processing Unit or GPU), as well as the abundance of data, led to the popularity of CNN-based architectures. There was way more training data than before, the networks could get much deeper (e.g. in terms of number of hidden layers), and the amount of time required for their training was dramatically decreased. All this in addition to the fact that CNN-based architectures started to outperform classical machine learning algorithms by significant margins, made Deep CNNs the state-of-the-art.

---

5This can actually be the case for any layer that receives its input through a kernel or receptive field, for example, pooling (subsampling) layers in LeNet-5.
for computer vision. In 2012 AlexNet [22] became the first CNN-based model to win the annual ImageNet ILSVRC Challenge [23]. Compared to LeNet-5, amongst other differences, it had more convolutional layers (and the three last convolutional layers immediately followed each other, without pooling layers in between). The number of feature maps for the layers were substantially higher as well.

The record-breaking success of AlexNet paved the way for the deeper, more complex CNN architectures. One particular approach to research for finding better architectures involves with stacking layers on top of each other, or coming up with innovative ways for connecting layers or neurons together. As an example, the winning model of the 2015 ImageNet challenge was ResNet [24], a CNN-based architecture that consisted of 152 layers and used residual learning and skip connections to largely increase the speed of training.

However, when researchers present their breakthroughs or claim their new models are better than the base models because of the modifications they have made, they usually fail to thoroughly motivate why these modifications are responsible for the claimed improvements. Following a well-known research practice for studying the brain, we now introduce the concept of Ablation Studies for machine learning, and Ablation Programming as the corresponding programming framework that enables machine learning practitioners to gain insight on the innerworkings of their machine learning systems.

2.2 Ablation Study

An Ablation Study, in medical and psychological research, is a research method in which the roles and functions of an organ, tissue, or any part of a living organism, is examined through its surgical removal and observing the behaviour of the organism in its absence. This method, also known as experimental ablation, was pioneered by the French physiologist Marie Jean Pierre Flourens in the early nineteenth century [25]. Flourens would perform ablative brain surgeries on animals, removing different parts of their nervous systems and observing the effects on their behaviour. This method has since been used in a variety of disciplines, but most prominently in medical and psychological research and neuroscience.

In the context of machine learning, we define ablation study as a sci-
entific examination of a machine learning system by removing its building blocks in order to gain insight on their effects on its overall performance. Dataset features and model components are notable examples of these building blocks (hence we use their corresponding terms of feature ablation and model ablation), but any design choice or module of the system may be included in an ablation study (see Figure 2.6 as an example). It should be noted that although an ablation study is of course usually not sufficient to draw conclusions on the contributions of different modules, when coupled with other empirical and statistical methods it can provide valuable insights to practitioners and researchers.

Figure 2.6: A machine learning system consists of several components.

Throughout the years, ablation studies have been included in a number of notable publications in machine learning research [26, 27, 28, 29, 30]. Much like the original ablation study in physiology, these experiments include removing specific features or specific components of the model from the training process, and observing the resulting performance. Also, following the increase in efforts towards explainable and interpretable machine learning systems, a number of recent works specifically discuss the role of ablation studies [31, 32].

In terms of tools and techniques, there have been efforts in developing frameworks and platforms that are focused on interpretability and explainability of machine learning models. These efforts constitute a wide spectrum of tools and techniques, from interactive visualization tools to software platforms and mathematical modeling frameworks and algorithms. Notable examples\(^6\) include SHAP [33, 34], LIME [35].

\[^6\]A list of tools and platforms for explaining machine learning systems can be found at

\[\text{https://github.com/EthicalML/awesome-production-machine-learning}\]
Despite these efforts, performing ablation studies is still neither a common practice nor part of the standard machine learning research methodology. We believe one of the main reasons for this is the fact that performing an ablation experiment requires specific frameworks or platforms, each with their own learning curves. Furthermore, the practitioners and researchers have to develop platform-specific code or make considerable modifications to their existing code, resulting in a huge burden in terms of time and effort.

Even if the practitioners undergo these extra efforts, when it comes to resource utilization and efficient and scalable computation, the state-of-the-art systems and tools for interpretability face major issues. On the rare occasion that they support distributed or parallel computation out-of-the-box, it is in a very basic and inefficient form, resulting in long-running experiments and/or low resource utilization of the expensive machine learning infrastructure. These challenges gravely disincentivize performing ablation experiments.

Now that we have presented the required theoretical background for reading this thesis, we will briefly discuss the systems, platforms, and frameworks that are related to our work.

### 2.3 Platforms and Frameworks

MAGGY [1] is an open-source Python framework for distributed, asynchronous computation on Apache Spark clusters, and includes APIs for running hyperparameter optimization and ablation studies. MAGGY has been developed to be used as part of the HopsML [41] ecosystem, so in this section we will have a quick introduction of Apache Spark, Keras, TensorFlow, and Hopsworks.

#### 2.3.1 Apache Spark

Apache Spark [2] is a general-purpose cluster computing framework that builds upon the MapReduce computing framework [42] by introducing in-memory computation abstractions of Resilient Distributed Datasets (RDDs) [43] and DataFrames [44]. These in-memory abstractions massively speed up computation compared to MapReduce and
its open-source implementation, Apache Hadoop MapReduce [45], primarily since they eliminate the need for (thus the overhead of) disk I/O for storing or retrieving intermediate results of a computation job.

Since becoming an Apache Software Foundation project in 2013, Spark has quickly become one of the most-popular frameworks for distributed computing. Written in Scala, it also provides rich APIs in Java, Python (known as PySpark), R, and SQL. Spark comes with a standard set of libraries, including Spark SQL for structured data, Spark Streaming for stream data processing, MLlib for large-scale machine learning, and GraphX for graph processing. MAGGY uses Spark as its back-end for distributed computing, and trials of an experiment run in parallel on Spark executors. However, since Spark follows the Bulk Synchronous Parallel execution model, parallel execution of trials might result in low resource utilization. We will discuss this issue in more detail in Section 3.1.

2.3.2 Keras and TensorFlow

Keras [6, 46] is a high-level open-source library for easy implementation of neural networks. An important feature of this library is that the same Keras code can be run on different backends, i.e. computing frameworks and platforms, including TensorFlow [4, 47]. Keras is regarded as one of the most-used machine learning libraries, and is also known for its clean and intuitive API design.

TensorFlow [4, 47] is an open-source library for dataflow programming, and a leading framework for machine learning and deep learning. Originally authored by the Google Brain team, a host of tools and libraries have been developed around TensorFlow to cover the whole machine learning pipeline, from research and development to production use. TensorFlow includes a number of APIs with different levels of abstractions, namely the Estimators API, the Functional API, and the Eager Execution API. TensorFlow also includes an implementation of Keras, and it seems that beginning with TensorFlow 2.0, TensorFlow’s Keras API is going to be its API-of-choice for development of machine learning and deep learning models.

Currently, MAGGY and its APIs are compatible with machine learning programs developed with TensorFlow’s Keras API.

\[\text{At the time of writing this thesis, TensorFlow 2.0 is scheduled to be released in the second half of 2019, but its Beta is already publicly available.}\]
2.3.3 Hopsworks

Hopsworks [5, 48] is an open-source managed platform for scalable and distributed machine learning and data science. It originated from a research effort to extend the scalability of the Hadoop Distributed File System (HDFS) [49], which led to the development of HopsFS [50, 51]. With the success of HopsFS, a managed platform named Hopsworks was built around it, allowing practitioners and researchers to develop data science and machine learning programs through Jupyter Notebooks and run them on a Spark cluster. Hopsworks also regards GPU as a resource in YARN [52], the resource manager in the Hadoop ecosystem. Figure 2.7 shows the platforms and tools that constitute the Hopsworks ecosystem.

![Figure 2.7: Hopsworks is a managed platform for performing scalable data science and machine learning.](image)

Hopsworks includes HopsML [41], a set of services and frameworks for developing machine learning pipelines in Python. A particular API of HopsML, the Experiment API, provides users with the ability to run their machine learning programs on a Spark cluster by just wrapping their machine learning code in a Python function in their Jupyter Notebooks. MAGGY has been designed and developed based on the Experiment API, to add support for asynchronous execution of programs and global control of machine learning experiments, which we will discuss in detail in the subsequent Chapters of this thesis.
2.4 Back to the Main Problem

As machine learning and deep learning systems grow in complexity and are used in more and more application domains, it becomes more essential that we possess the ability to explain and interpret their inner workings. It is also equally important that we exercise this ability and make interpretability experiments as part of the standard machine learning research methodology. To this end, it is crucial that the corresponding tools and frameworks be usable and do not add a significant burden to the already complex machine learning practice, since after all the main goal in machine learning research is to improve the capabilities and performance of the models, algorithms, and systems.

We believe the Ablation API we have developed as part of MAGGY is a significant step in this path, as it enables machine learning researchers and practitioners to perform ablation studies with very low modifications to their existing code, with very little effort. Also, the execution model of MAGGY allows for higher resource utilization compared to running the trials in parallel on Apache Spark clusters.

The rest of this thesis discusses the design and implementation of MAGGY and its Ablation API.
Chapter 3

Design and Implementation

The contributions of this thesis include the design and development of MAGGY, an open-source Python framework for asynchronous computation on Spark clusters, and an API for parallel ablation studies using MAGGY. Our efforts were primarily focused on enabling the support for early-stopping of machine learning trials that run on Spark executors without the need for relaunching the whole Spark application or dealing with barriers, as this has a devastating effect on resource utilization. Once we developed our solution, we were able to develop APIs that could efficiently parallelize steps in a typical machine learning workflow. In particular, we focused on hyperparameter optimization and ablation studies, and in this Chapter we discuss the design and development of MAGGY and its API for parallel ablation studies.

3.1 Requirements, Assumptions, and Goals

The concept of iteration is inherent in the machine learning practice, and in different levels of abstraction. For example, optimizing the parameters of a machine learning model using the back-propagation algorithm \cite{54} requires several passes over the data (also known as epochs). As another example, consider the search for better hyperparameter values, which usually means that the same base model is trained with different combinations of hyperparameters, resulting in different final models. The first example deals with iterations over the data, where the same model eventually gets better and better. However, in the second example, we have a higher level iteration that results in several models, where for training each model we have to iterate over the data.
Many of these iterations, it turns out, are easily parallelizable. The standard practice of hyperparameter optimization includes running an experiment in which several sets of values for the hyperparameters are evaluated. Each set of values (a specific combination of values of different hyperparameters), gets evaluated in one trial. A trial, in turn, consists of training and evaluating a machine learning model given a set of hyperparameter values.

A similar thing holds for an ablation study, as depicted in Figure 3.1. Here, our experiment is the ablation study itself, and each model ablation trial involves training a model with one or more of its components removed. Similarly, a feature ablation trial involves training a model using a different set of dataset features, and observing the outcomes.

![Figure 3.1: Examples of ablation trials: a feature ablation trial (top) and a model (layer) ablation trial (bottom).](image)

### 3.1.1 Framework General Requirements

Apache Spark is one of the most popular data intensive computing platforms, and Spark clusters are being used by many for various steps in their machine learning pipelines. Now, if we want to parallelize an experiment using an Apache Spark cluster, our experiment would be represented as an Spark application, and each trial (hyperparameter optimization or ablation) would be represented as an Apache Spark task. In the Bulk Synchronous Parallel processing model that Spark is based on, starting a new set of trials requires all the previous trials in a stage to finish first, therefore introducing a barrier. For hyperparameter optimization this means that although we can soon realize that a trial
is promising or not, we have to allocate all our resources to all the trials to execute completely, irrespective of the fact that they are good trials or not.

Figure 3.2 shows how executing an experiment in a distributed manner on a Spark cluster works. We can distinguish between undirected and directed experimentation, where the former does not require a global decision making component, but the latter depends on results from previous trials. The barriers caused by the execution model of Spark means that all trials in a stage need to be finished before we can start new trials, so if there are trials that are finished sooner than the others because of the trial-specific settings or the execution environment, those executors would sit idle until all the other trials on the other executors are finished. Also, in this case there is no easy way for early-stopping non-promising trials, further contributing to under-utilization of the resources.

Based on these assumptions, our primary requirement for an efficient distributed framework is to have a global experiment controller. The immediate gain is trivial in the context of hyperparameter optimization: the controller can propose new trials based on the results of the latest finished trials, a pattern that fits perfectly with approaches such as Bayesian optimization. For ablation studies, this can also hold and it enables us to develop ablation study policies that exploit the results of the latest finished trials. Another added benefit is that we will be able to schedule the trials in a more intelligent manner, for example by developing a trial planner for an ablation study, that plans trials that will be finished sooner with a higher priority, or eliminates upcoming
trials that are assumed to be useless based on the current results, while the experiment is already running.

### 3.1.2 Specific Requirements for the Ablation API

The end-users of our APIs are machine learning practitioners and researchers, who are familiar with the programming models of popular frameworks such as TensorFlow or PyTorch, with different levels of experience. Hence, in order to persuade the end-users to include ablation studies in their research, this inclusion should be possible with as little effort as possible, be it in terms of developing specific code for the ablation study, or learning about how to develop this code. Simply put, we want the users to perform ablation studies with minimal change to their existing machine learning code. This has another important benefit: almost the same code can be put to production, and there is no extra step for translating or modifying the code to the production setting. Therefore, whenever possible, we prefer code extension to code modification.

To this end, we exploit best-practices and common idioms that are used within these popular frameworks. One such idiom is defining the machine learning or deep learning model in form of a Python function. Also, we are manipulating the models, or the datasets used for training, so we use relevant techniques and practices in software engineering and metaprogramming \[55\], namely reflection, dependency injection, and function generation.

### 3.1.3 Summary of Requirements

To put it all together, our desirable framework should fulfill the following requirements:

- **Global experiment controller.** Leveraging the data and metrics obtained by evaluating individual trials of the experiment requires a component that is able to access and aggregate this data, and modify the existing experiment either by generating new trials, or eliminating existing scheduled trials. This component, however, must not become a bottleneck itself, so it should follow the asynchronous computation paradigms.

- **Support for early-stopping of arbitrary trials.** Effective application of a global experiment controller implies that we also be
able to stop arbitrary trials and assign new trials to the execu-
tors without interrupting the whole experiment or affecting other executors.

• **Machine Learning platform agnosticism.** Core functionality as well as end-user and developer APIs of the framework should be designed in a way that is independent of the underlying machine learning platforms such as TensorFlow and PyTorch. Although we exploit common idioms and best-practices that are associated with popular frameworks, it should be possible to extend the framework to support other standard Python-based frameworks without a significant development burden.

• **Fault-tolerance.** Failure is intrinsic to any distributed or parallel computing setting. Especially in the case of large machine learning and deep learning models, training a single model, which means executing one trial in our setting, can take hours or even days. Hence, the framework should be resilient to failures of arbitrary worker nodes. In other words, if something goes wrong during the execution of a trial, it should not bring down the whole experiment or affect other workers that are progressing without problem.

• **Easy extendibility.** Apart from the APIs that are used by the end-users, the framework should also expose APIs for core developers so that they can further extend the framework or implement their own experiment control policies, e.g., different hyperparameter optimization or ablation policies. In the case of implementing new control policies, it should be possible without getting involved with the particularities of the distributed computation and the underlying Spark engine.

We now discuss the design and implementation of MAGGY, as the core framework, and the Ablation API, and see how they satisfy the aforementioned requirements.

### 3.2 MAGGY Core

We first review the implementation of MAGGY, our Python-based distributed asynchronous computation framework on top of Apache Spark.
We start by going through its core architecture, and see how it integrates with Apache Spark and Hopworks, and then discuss the APIs that are exposed to the end-users and developers.

**Architectural Overview**

To enable support for global experiment controllers and early-stopping of trials, our most crucial requirements, MAGGY uses an RPC framework. This enables executors to communicate with the driver. Figure 3.3 shows a high-level view of core MAGGY components.

![High-level view of the architecture of MAGGY.](image)

**MAGGY API**

This is the Python API for the developers and end-users of MAGGY, and as of now it consists of the Hyperparameter Optimization API and the Ablation API. Users can define their hyperparameter optimization or
ablation study experiments, and launch them through this interface. We will cover the Ablation API of MAGGY in detail in Section 3.3.4.

Components Running on the Spark Driver

The ExperimentDriver, running within the Spark Driver, is the high-level abstraction that takes care of scheduling trials, and is the global experiment controller that we have been talking about. It hosts an experiment-type-based controller, e.g., an optimizer for hyperparameter optimization, or an ablator for ablation studies. It also manages the RPC Server thread, and receives periodic trial metrics and status messages from the RPC Clients that run on executors, and writes them to a message queue. The worker thread of the ExperimentDriver is the heart of the controller: it constantly polls the queue, and acts according to the type of the messages or the status of the experiment. Among its other roles, the worker thread is responsible for initiating the check for early-stopping, interacting with the optimizer or the ablator in order to get new trials, and assigning new trials to the executors. The worker thread communicates with the RPC Server through a shared data structure, for keeping track of executor-related and trial-related data.

Components Running on Spark Executors

The actual trials run on Spark executors. The users just have to wrap their training code in a Python function, and the function will be serialized and sent to the executors once the Spark application is launched. Each executor also runs an RPC Client thread that communicates with the RPC Server running on the ExperimentDriver. The task starts by setting up the client-server communication and launching the RPC Client thread. After this, the rest of the task is nothing but a while loop that executes the wrapped training function by passing it the trial-specific parameters, reports the metrics and status to the ExperimentDriver, and asks for a new trial if the current trial is finished or interrupted (e.g., early-stopped). This while loop, and consequently the task itself, is finished once there are no more trials to be assigned to the executor.
**Interaction with Hopsworks Platform**

**MAGGY** uses HopsFS \[50\] as the persistent storage layer, for both accessing the datasets as well as logging and checkpointing. The users can develop, launch, and track the progress of their machine learning applications in Jupyter Notebooks. **MAGGY** also writes logs and JSON specifications of the experiments to Elasticsearch \[56\], and the users can also track or search through previous experiments from the Experiments dashboard in the Hopsworks web UI \[5\].

**Trial Representation**

In **MAGGY**, each new trial is passed to the executors in terms of a dictionary of the parameters that are necessary to execute a specific trial. For a hyperparameter optimization experiment, the keys of this dictionary include the hyperparameters and their trial-specific values. For an ablation trial, as we will see in the following Section, this dictionary includes two Python callables: a dataset generator function, and a model generator function. These parameters also should be defined as the parameters of the wrapped of training function.

### 3.3 Ablation on MAGGY

Now we present the main contribution of this thesis, an API for ablation study of machine learning systems on top of **MAGGY**. We will first introduce the notion of an ablation policy, which determines how the machine learning system - the model, the dataset, or the other components - should be changed, or in this context, ablated. We also present LOCO, a general ablation policy that we have implemented to demonstrate the API. We then explain how the ablation study is realized under-the-hood, and discuss the User API and Developer API of **MAGGY** Ablation.

#### 3.3.1 Ablation Policies

An ablation policy specifies how the components of a machine learning system should be added, removed, or altered in an ablation study. In **MAGGY**, an ablation policy is implemented in terms of an ablator, which is used by the ExperimentDriver (the global experiment controller) to generate the trials of the ablation study. Just like different
hyperparameter optimization algorithms, MAGGY allows developers to easily implement their own ablation policies, details of which will be discussed in Section 3.3.5. However, we have implemented an ablation policy that we believe might be the first policy that comes into mind when someone thinks about an ablation study. We have named it LOCO, which stands for Leave One Component Out.

**LOCO: Leave One Component Out**

If we consider a machine learning program as a system, it is made of a number of components (that could be made of smaller components themselves). For example, the dataset that we train or test our machine learning model is a high-level component of the system, which itself could be expressed e.g. in terms of several features or columns. As another example, a deep neural network consists of several hidden layers, but also these layers could be connected to each other in different patterns, e.g. using dropout [57]. Hence, we consider every property or design decision of the machine learning system as a potential component. A machine learning practitioner or researcher who wants to gain more insights about the inner workings of their developed system, in turn, would naturally be interested of knowing how the performance of the system might change if each of these components, or groups of components, are removed.

Therefore, the most natural ablation policy might be the one that involves removing components from the system, one at a time. We call this the *LOCO* policy (short for *Leave One Feature Out*). The current implementation includes feature ablation as well as model ablation.

**Feature Ablation**

The goal of a feature ablation experiment is to understand the effects of the different features of the dataset, on the performance of the system. To this end, one can train and evaluate their machine learning model with different variations of the dataset that only differ in their included features. This is exactly what feature ablation in *LOCO* is about.

---

1The inspiration for this name comes from LOFO Importance [37], where they only deal with dataset features.
Model Ablation

Similar to feature ablation, a *model ablation* experiment involves removing - or changing - components of the model and train and evaluate the system using these variations of the model. The differences in results can then be further studied e.g. to find correlations or causal relationships. MAGGY’s implementation of LOCO currently provides support for removing layers of a neural network through a simple API. However, it can easily be extended to support other components of the model, due to the way it parses and manipulates the model object, as we will discuss in the next Section.

3.3.2 Implementing MAGGY Ablation

The Ablation API of MAGGY has been implemented using Python 3. The AblationStudy class is used to specify the components that the users want to include in their ablation experiment, such as dataset features or model layers. However, it does not include the ablation policy, as one might want to try different ablation policies with the same components of the machine learning system.

The ablation policies are implemented by extending a class called AbstractAblator. This extended class, or ablator, is responsible for generating the trials, and is instantiated by the ExperimentDriver. It should contain two functions that return callables, for generating the datasets, and generating the models. The ablator should also implement an initialize() method that generates a number of Trial objects before the actual experiment is started. For example, in the case of LOCO, since no new trials will be generated during the experiment, it makes sense to generate all of them beforehand. However, there might be policies that dynamically add, remove, or alter trials to the experiment. In such cases it is still mandatory to implement this initialize() method so that enough trials are available for the experiment to start.

The ablator stores to-be-executed Trial objects in a Python list, and the ExperimentDriver can request a new trial by calling the get_trial() method of the ablator. This method also accepts an optional trial parameter, which the ablator can use to generate a new trial based on its information. This allows for dynamic trial generation and scheduling. In any case, this method should return a Trial object, and the ExperimentDriver will assign a new trial to a Spark executor
using this object.

On the Spark executor side, in the while loop of the long-running task, the RPC Client makes a blocking call to receive serialized messages from a socket that the RPC Server and Clients use to communicate with each other. Once a message is received, it is de-serialized and if it contains a new trial, the parameters of the trial (the dataset generator function and the model generator function) are extracted from the Trial object in the message and passed to the training function that was wrapped by the user. Then this wrapped function, that includes the whole training loop, will get executed, and metrics are broadcasted to the ExperimentDriver through heartbeats.

Next, we will discuss the implementation of the LOCO ablation policy, and how we manipulate the dataset and the model and create the corresponding generator callables.

3.3.3 Implementing the LOCO Policy

LOCO is a general ablation policy, so it should create trials for both feature and model ablation. Also, this policy also implies a fixed number of trials, and these trials can be created all together in the initialize method, appended to a list, called trial_buffer. This means that any invocation of the LOCO ablator’s get_trial() method by the ExperimentDriver simply returns a Trial instance from trial_buffer.

Feature Ablation

The LOCO ablator of MAGGY currently supports feature ablation for datasets that are generated from files stored as TFRecord, TensorFlow’s format for storing sequences of binary records [58]. It also uses the Feature Store in Hopsworks [59] to access dataset metadata, including the path to dataset files, the names and types of features, and the schema of the dataset. It then reads the TFRecord files, and for each feature that is specified to be included in the feature ablation study, it creates a callable that upon invocation, returns a TFRecordDataset that does not contain that feature. This implementation follows best-practices for creating and working with data in TensorFlow [58, 60, 20].

An important point to notice here is that we want the call for creating the TensorFlow dataset to be made on the Spark executors, as they have access to the dataset files via the distributed file system (HopsFS).
Otherwise, we would have to ship custom datasets with each trial, which would be very inefficient and in complete contradiction with the MapReduce programming model. This is why the Trial objects contain a Python callable for creating the dataset.

Model Ablation

Currently, MAGGY supports ablation studies for layers of Keras models. It uses a method of the tf.keras.Model class of TensorFlow’s Keras API [61], called get_config(). This method returns a Python dict that contains human-readable information about the structure of the model. The value of the key called ‘layers’ includes a Python list, and each item of this list represents a layer and its configuration, in form of dict. So, just to recap, we are dealing with a dict that has a list as the value of one its keys, and this list is made of dicts. One particular key of these inner dictionaries dict is ‘name’. This is the name the user can give to the layer when defining the model through the Keras API, and this is how we identify the layers to be ablated from the model.

So, an ablator with the LOCO policy receives a base model generator that the user should define (a Python callable that returns a tf.keras.Model) and pass it as a parameter to the user-wrapped function. Then, for each layer or groups of layers that are included in the ablation study specification, it uses that generator to create a model, retrieve its configuration using get_config(), change the configuration (remove the layer or a group of layers), and pass the modified configuration to tf.keras.models.model_from_config(). These models are then returned as the return values of the model generator callables, and corresponding trials will be created with these callables.

3.3.4 User API

Using the Ablation API of MAGGY consists of three general steps, as shown in Figure 3.4. The first step is for setting up the ablation study experiment, using MAGGY’s Ablation API. The second step is where the user develops the training code using their platform of choice [6] and

---

2Currently we support code developed with the Keras API of TensorFlow, but full support for other popular platforms will be gradually added to MAGGY.
wraps it in a Python function. After that, as the third and last step, the user will initiate the parallel execution of the experiment by providing the wrapped function and experiment settings.

![MAGGY Ablation User API workflow](image)

**Figure 3.4:** MAGGY Ablation User API workflow.

Setting up the experiment starts with creating a new `AblationStudy` instance. The user has to provide the name of the base dataset in the feature store of the project, as well as the name of the label. Optionally, the user can specify the version of the dataset, in case there are multiple versions of the same dataset in the feature store. An example is shown in Listing 1 where we use the well-known Titanic Dataset [62].

```python
import tensorflow as tf
import maggy
from maggy.ablation import AblationStudy

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

**Listing 1:** Creating an `AblationStudy` instance.

Next, the user can add arbitrary components to the ablation study. For this, we use the term `include`. With the current implementation, the user can include `features`, `layers`, and `layer groups` in the ablation study. Especially, `layer groups` can be used for very deep models that include blocks or groups of layers with similar configuration. A common practice in developing code for these models is to generate this blocks of layers using e.g. a `for` loop, and giving prefixes to the layer names.
Example usage of the API for including components in the ablation study is shown in Listing 2.

```python
# include features
ablation_study.features.include('pclass', 'fare')

# include layers
ablation_study.model.layers.include('my_dense_two', 'my_dense_three')

# add a layer group using a list
ablation_study.model.layers.include_groups(['my_dense_two', 'my_dense_three'])

# add a layer group using a prefix
ablation_study.model.layers.include_groups(prefix='my_dense')
```

Listing 2: Adding components to the ablation study.

The user also has to provide a base model generator function. This function should return a `tf.keras.Model` [61]. Here, the user can also provide a name for the layers. Listing 3 contains an example of a base model generator for a simple Sequential model, and its inclusion in the study.

```python
def base_model_generator():
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Dense(64, activation='relu'))
    model.add(tf.keras.layers.Dense(64,
        name='my_dense_two',
        activation='relu'))

    model.add(tf.keras.layers.Dense(32, activation='relu'))
    model.add(tf.keras.layers.Dense(2,
        name='my_dense_two',
        activation='sigmoid'))

    # output layer
    model.add(tf.keras.layers.Dense(1, activation='linear'))
    return model

# include the generator function in the study
ablation_study.model.set_base_model_generator(base_model_generator)
```

Listing 3: Defining the base model generator.
Now for the second step of the workflow, the user has to wrap their machine learning code (e.g. training code) in a Python function. The function has three parameters. The first two, `dataset_function` and `model_function`, should be used in the body of the function as callables for creating models and datasets. Each trial of MAGGY contains these callables, and depending on the trial configuration, these callables will then generate the corresponding model and dataset. A third parameter, `reporter`, can be used for reporting metrics from the executors to the driver with heartbeats. The users can use callbacks based on the ML platform they are using. Currently, MAGGY supports Keras callbacks. Listing 4 demonstrates this step.

```python
from maggy import experiment
from maggy.callbacks import KerasBatchEnd

def training_function(dataset_function, model_function, reporter):
    import tensorflow as tf
    epochs = 5
    batch_size = 10
    tf_dataset = dataset_function(epochs, batch_size)
    model = model_function()
    model.compile(optimizer=tf.train.AdamOptimizer(0.001),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    ### Maggy REPORTER
    callbacks = [KerasBatchEnd(reporter, metric='acc')]

    history = model.fit(tf_dataset,
                        epochs=5,
                        steps_per_epoch=30)
    return float(history.history['acc'][-1])
```

Listing 4: Wrapping the user code in a function.

Finally, the user can launch the experiment by passing the wrapped function and some other experiment-specific parameters to `maggy.experiment.lagom()`[^3] and view the progress and the result of the experiment in the same Jupyter Notebook cell.

[^3]: Lagom is a Swedish word that means just the right amount.
result = experiment.lagom(map_fun=training_function,
    experiment_type='ablation',
    ablation_study=ablation_study,
    ablator='loco',
    name='Titanic-LOCO',
    hb_interval=5)

Listing 5: Launching the experiment.

### 3.3.5 Developer API

Among other things, developers can implement their own ablation policies by extending the *AbstractAblator* class. The *AblationStudy* instance that the user sets up is passed to the *init_*() function of *AbstractAblator*, and this abstract class includes a number of abstract methods that provide flexibility for implementing custom ablation policies. These methods are shown in Listing 6.

```python
@abstractmethod
def get_number_of_trials(self):
    pass

@abstractmethod
def get_dataset_generator(self, ablated_feature=None,
    dataset_type='tfrecord'):
    pass

@abstractmethod
def get_model_generator(self, ablated_layer):
    pass

@abstractmethod
def initialize(self):
    pass

@abstractmethod
def get_trial(self, ablation_trial=None):
    pass

@abstractmethod
def finalize_experiment(self, trials):
    pass
```

Listing 6: Abstract methods for implementing custom ablators.
The first method, `get_number_of_trials()`, can be used e.g. by the driver to know how many executors it should assign to the experiment. In many ablation policies such as LOCO, we already know how many trials will be in the experiment before its start, so we might be able to leverage this knowledge for scheduling trials. Also, for dynamic ablation policies, where new trials might be generated while the experiment is running, calls to this method can be used to control the experiment.

The second and third abstract methods of `AbstractAblator`, `get_dataset_generator()` and `get_model_generator()`, are the key parts of the ablation policy. These two methods should return Python callables, that will then get passed to the user wrapped code. As an example, recall that during setting up the ablation study, the user has to define a base model generator and include it in an `AblationStudy` instance. Since the `AblationStudy` instance is passed to the `__init__()` function of `AbstractAblator`, an implementation `get_model_generator()` can then make modified versions of the base model and return callables for generating the modified models.

The `initialize()` and `finalize_experiment()` methods are called in the beginning and the end of the experiment, respectively. For static ablation policies such as LOCO, all the trials may be generated in `initialize()`, but it can also be the case that only a limited number of trials are generated and the rest will be created as the experiment is making progress. Once an experiment is finished, the Worker thread of `ExperimentDriver` makes a call to `get_trial()` to get a new trial if there exists any. The `ExperimentDriver` can also pass a finished trial instance when calling this method, and that can be used for returning or generating new trials based on the results of the finished trials. `finalize_experiment()` can be used for cleaning up and handling resources of the ablator before the experiment is finished.
Chapter 4

Results and Discussion

In this Chapter, we briefly discuss how the design and development of MAGGY and its Ablation API address the key requirements of this project, as mentioned in Sections 1.3 and 3.1. We also compare our APIs with two well-known frameworks for system experimentation: keras-tuner [63] and LOFO Importance [37].

4.1 Addressing the Goals and Requirements

The architecture of MAGGY, as discussed in Chapter 3 and shown in Figure 3.3, satisfies our most crucial requirement by providing a global experiment controller through the ExperimentDriver running on the Spark Driver, and the Early-stop Check thread within its Worker thread. As an immediate result of this design, new trials can be generated based on the results of the previously finished (or stopped) trials, and they can be started without the need for waiting until all the running trials are finished.

Also, the core functionality and the programming model of MAGGY is not bound to any specific machine learning platform. The simple function-wrapping programming model allows for parallel execution of trials within experiments, given that the code is wrapped inside a Python function and returns a metric - or value - that may be used for generating new trials, optimizing a parameter (hyperparameter tuning), or comparing the performance of different variations of the system (ablation study). The Developer API of MAGGY allows for building up on its functionality by implementing controllers for different tasks. MAGGY, however, can still leverage platform-specific features where de-
sirable. An example of these could be the current support for Keras callbacks.

Fault-tolerance in MAGGY is provided both by check-pointing the intermediate results to HopsFS in form of human-readable JSONs, and by Apache Spark itself, on a lower level. However, this is still an area that can be improved. Similar to fault-tolerance is the concept of reusability, where we might be able to reuse intermediate results of the experiment, e.g. materialized model configurations from previous ablation trials, to avoid redundancy.

Finally, regarding ablation studies, our main goal was to design an API that can enable machine learning researchers and practitioners to include ablation studies in their experiments with minimal effort. To this end, we have leveraged best-practices for developing machine learning code, and the resulting API requires very few changes to the existing code (e.g. wrapping the code in a Python function, and adding a name parameter to the layer constructors when defining the model).

### 4.2 Comparison with Other Frameworks

Now, we compare MAGGY with two well-known frameworks that are used for hyperparameter tuning (keras-tuner) and feature ablation (LOFO Importance).

keras-tuner is a hyperparameter tuning framework developed and maintained by the team behind the Keras framework (keras-team). It has been recently released, and uses a workflow that is similar to MAGGY’s user API. The most important similarity is the fact that the users are required to wrap their model definition code in a function. The users can then launch an experiment using a number of hyperparameter optimization algorithms, and retrieve the best performing models. However, the framework is not parallelized by design, it cannot leverage global knowledge of the experiment, it does not support early-stopping, and it can only be used for hyperparameter tuning.

LOFO Importance is a framework for calculating the importance of the features of a dataset when used for training a machine learning model. As the name suggests, it only supports feature ablation, but it runs multiple trials in parallel, using Python’s standard multiprocessing package.

Table 4.1 is a summary of this comparison. Compared to the other
Table 4.1: Comparing MAGGY with keras-tuner and LOFO Importance

<table>
<thead>
<tr>
<th>Framework</th>
<th>Global Controller</th>
<th>Early-Stopping</th>
<th>Parallel by Design</th>
<th>Unified API</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAGGY</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>keras-tuner</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>LOFO Importance</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

two frameworks, MAGGY provides global control of the experiment, support for early-stopping, and is parallel by design since it distributes the trials to be run on an Apache Spark cluster. Furthermore, MAGGY includes a unified API for both hyperparameter optimization and ablation studies, so the users can perform these two experiments using the same programming model and the same API structure, with very few modifications to their existing code.
Chapter 5

Conclusion and Future Work

In this final Chapter of the thesis, we will summarize the work, and provide directions for the future development of MAGGY and its APIs.

5.1 Conclusion

We discussed the problems that arise with the increased usage of machine learning and deep neural networks. The outlook of the machine learning and deep learning research implies that much of the future research will focus on machine learning models that are more complex and deeper, and are trained on much larger datasets. This means that we need scalable platforms and frameworks that can efficiently train these growing-in-size models on growing-in-size datasets, and efficiently find the near-optimal hyperparameter combinations. On the other hand, because of this increase in complexity and the fact that machine learning systems are being used in more and more applications of our everyday lives, it is of the utmost importance that we gain the capability of easily interpreting our machine learning systems. We need to be able to open the so-called black box of deep learning.

On a more technical level, we saw the problems that arise by distributing the trials of hyperparameter optimization and ablation studies using the traditional Spark programming framework. Spark’s bulk synchronous parallel model means that the efficiency and resource utilization of the expensive deep learning clusters would not be satisfactory, due to the wasted compute and time spent for under-performing models or unpromising hyperparameter combinations. We also explained the need for a global experiment controller, so we can efficiently lever-
age the information obtained from the latest finished or stopped trials during an experiment. Another point of interest was to see if we could unify hyperparameter optimization and ablation studies, as both of these tasks involve with making a particular change in the model but repeating a fixed set of steps for that modified version, which is by its nature embarrassingly parallelizable.

We addressed these requirements by introducing MAGGY, a Python-based, open-source asynchronous computing framework on top of Apache Spark and Hopsworks. We saw that using an RPC-based framework and by exploiting the threading mechanism of Python, we can establish efficient asynchronous communication between the Spark driver and a set of Spark executors. We implemented the support for different early-stopping policies, and developed APIs for hyperparameter optimization and ablation studies. The API for ablation studies was the particular focus of this thesis, and we explained how the trials are created, executed, and how the users can add ablation study experiments to their machine learning code without much effort or the need for making considerable modifications in the code.

5.2 Future work

MAGGY is an on-going research and development project, and welcomes contributors from different backgrounds to improve or add to its functionality, extend its applications and usability, or audit and assess its performance. However, we think the following are good directions for future research and development.

5.2.1 Core MAGGY Platform

Add Support for More Platforms

Obviously, not every researcher or practitioner uses TensorFlow or Keras, and recently there has been an increase in the popularity of the PyTorch [64] platform. Since MAGGY has been designed with platform-agnosticism in mind, it should not be hard to add the support for other popular machine learning and deep learning frameworks. This, however, requires that one acquires a sound understanding and expertise with the core architecture as well as the high-level APIs of these platforms.
Dynamic Interaction with Trial Configurations

By this we mean the ability to interfere with the experiment, while it is being executed as an Spark application, to dynamically add or change the hyperparameter values or ablation trial configurations. Of course, this interference must not cause the Spark job to stop or fail. One way for doing this can be to develop an interface for the communication of the global experiment controller (optimizer or ablator) with the execution environment (e.g. the Jupyter notebook).

Straggler Take-over

Current implementation of MAGGY is already resistant to stragglers (under-performing or slow worker nodes). However, if all the executors have finished their trials with the exception of one or a few straggler nodes, it might be the case that the total experiment time (a.k.a. the makespan) is hugely affected because of these exceptional straggler nodes. It is beneficial, and not so hard, to implement a mechanism that assigns duplicates of the straggling trials to the idle executors. These executors could in fact finish the trial sooner than the stragglers, and the whole experiment would in turn end faster.

5.2.2 MAGGY Ablation

More Complex Ablation Policies

As the most natural ablation policy, and to demonstrate the concept of ablation studies, we have implemented the LOCO ablation policy. However, the fact that we provide the AbstractAblator class for developers to build up on, invites for the implementation of a diverse set of ablation policies. This can also include dynamic ablation policies, by which we mean ablaters that can generate new trials or modify existing ones based on the results from previously finished trials.

Non-blocking Ablator Initialization

A key assumption we had in designing the global experiment controller was that the time it takes for generating a new trial is negligible compared to the time it takes for execution of that trial. In the current implementation, the ablator has to be initialized before the Spark tasks are started on the executors. This initialization, of course, provides the
trials that are need for the tasks; however, in many ablation policies such as LOCO, the total number of trials can far exceed the number of available executors. The defining effect of this blocking behaviour becomes more obvious if we consider generating ablation trials for very large models or very large datasets. A good, truly asynchronous approach, can only block the driver program until a sufficient number of trials are generated, and initiate the execution of those trials on the executors. While the executors are busy with their assigned trials, the ablator can generate the remaining trials in the background.

**Trial Planner**

Following the discussion on the blocking behavior of the current ablator, it would be interesting to develop a trial planner that first generates and assigns trials that can be executed sooner. Especially, this would be truly effective if one is using a dynamic ablation policy.

**Support for Different Dataset Formats**

Again, similar to the case of different platforms, the datasets might come in different formats. TFRecord is TensorFlow's recommended - and own - format for dataset files, however there are other formats that are popular as well. In fact, file formats that allow columnar storage and access, such as Apache Parquet [65] are an excellent fit for feature ablation as opposed to the sequential binary format that is TFRecord. Hence, Providing support for these data formats would be a great improvement.

**Leverage Other APIs of TensorFlow**

The current work only targets the Keras API of TensorFlow. Keras is becoming the de-facto standard API of TensorFlow, but TensorFlow also includes a range of low-level and high-level APIs that each can provide more potential for richer ablation policies and execution plans. For example, it would be interesting to exercise the capabilities of APIs and classes such as Graph (tf.Graph), Estimator tf.estimator, and Functional, as well as TensorFlow's native support for custom models and layers.
Coming Up With, and Supporting Other Types of Experiments

Last but not least, MAGGY now supports two types of experiments related to machine learning and deep learning: hyperparameter optimization, and ablation studies. It would then make sense to think about other experiments that we can do with or on a machine learning system. One idea would be to investigate mixed precision training. For example, for the same model and dataset, it might be interesting to try different configurations of precision during training, and compare the performance of the resulting models.
Bibliography


[60] Importing Data - TensorFlow Core. URL: https://www.tensorflow.org/api_docs/python/tf/keras/Model.


