AutoAblation: Automated Parallel Ablation Studies for Deep Learning

Sina Sheikholeslami
KTH Royal Institute of Technology
sinash@kth.se

Moritz Meister
Logical Clocks AB
moritz@logicalclocks.com

Tianze Wang
KTH Royal Institute of Technology
tianzew@kth.se

Amir H. Payberah
KTH Royal Institute of Technology
payberah@kth.se

Vladimir Vlassov
KTH Royal Institute of Technology
vladv@kth.se

Jim Dowling
KTH Royal Institute of Technology
Logical Clocks AB
jim@logicalclocks.com

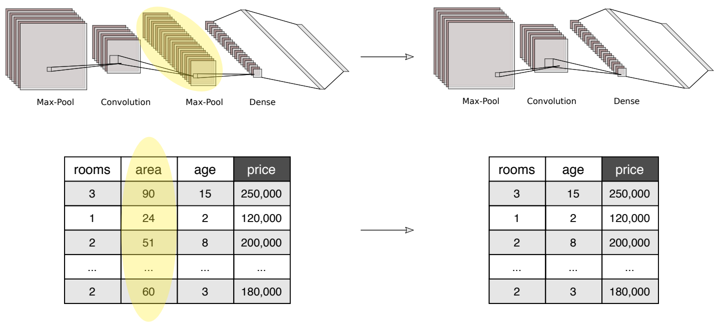
Abstract

Ablation studies provide insights into the relative contribution of different architectural and regularization components to machine learning models' performance. In this paper, we introduce AutoAblation, a new framework for the design and parallel execution of ablation experiments. AutoAblation provides a declarative approach to defining ablation experiments on model architectures and training datasets, and enables the parallel execution of ablation trials. This reduces the execution time and allows more comprehensive experiments by exploiting larger amounts of computational resources. We show that AutoAblation can provide near-linear scalability by performing an ablation study on the modules of the Inception-v3 network trained on the TenGeoPSAR dataset.

CCS Concepts: • **Computing methodologies** → **Model development and analysis**; **Machine learning**.

Keywords: Ablation Studies, Deep Learning, Feature Ablation, Model Ablation, Parallel Trial Execution

ACM Reference Format:
Sina Sheikholeslami, Moritz Meister, Tianze Wang, Amir H. Payberah, Vladimir Vlassov, and Jim Dowling. 2021. AutoAblation: Automated Parallel Ablation Studies for Deep Learning. In *The 1st Workshop on Machine Learning and Systems (EuroMLSys'21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3437984.3458834



rooms	area	age	price
3	90	15	250,000
1	24	2	120,000
2	51	8	200,000
...	...	...	...
2	60	3	180,000

rooms	age	price
3	15	250,000
1	2	120,000
2	8	200,000
...	...	...
2	3	180,000

Figure 1. Example trial configurations for model ablation (up) and feature ablation (bottom). Yellow highlighting indicates a "component" that is removed for the trial.

1 Introduction

Inspired by how the mammalian brain works, Deep Neural Networks (DNNs) have been at the forefront of recent breakthroughs in Artificial Intelligence. Since the early 19th century, a surgical procedure called an *ablation study* has been developed to understand the role of different components of the brain [3]. An ablation study involves removing a specific part of the brain of a mammal, and observing any resulting changes in its behavior. Given our limited understanding of brain function, this black-box approach has helped identify regions in the neocortex that are specialized for controlling specific behaviours and the relative contribution of brain regions to global function.

Similar to brains, we lack models to understand the function of Deep Learning (DL) systems at both the macro and micro levels. As such, black-box experiments that modify model architectures while observing system performance offer an approach to help improve our understanding of a DL system. An ablation study in DL involves measuring the performance of a network after removing one or more of its components to help understand the relative contribution of the ablated components to overall performance [24]. Dataset features and model components (e.g., layers) are notable examples of ablatable components, but any design choice or module of the system can be considered in an ablation study.

In this setting, we consider the execution of an ablation study on a given DL model or dataset as a single *experiment* consisting of several *trials*, where each trial involves removing one or more model or dataset building blocks, e.g., layers or features. We distinguish two kinds of ablation trials, namely, *model ablation* trials and *feature ablation* trials, depending on what kind of block is ablated in the trial: model components (e.g., layers or modules), or dataset features, respectively. Each model ablation trial involves training and evaluating a model with one or more of its components (e.g., layers) removed (Figure 1, (up)). Similarly, a feature ablation trial involves training and evaluating the model using a different subset of features in the dataset (Figure 1, (bottom)).

Over the years, many Machine Learning (ML) papers have included ablation studies [2, 9–11, 22]. Moreover, following the recent trend towards explainable and interpretable ML systems, several recent works [13, 17], discuss how ablation studies can lead to more explainable ML models. However, it can be observed that a significant part of the ML research community still regards performing ablation studies to be unnecessary, and when researchers publish or propose new model architectures or training procedures, they may attribute the resulting gains only to the changes they have made to a base model architecture or training procedure, without performing any ablation studies that would allow to identify and quantify the actual impact of each of these proposed changes [13].

Based on our observations, two main reasons for this oversight are: (i) performing ablation studies requires maintaining redundant copies of code that each correspond to a different configuration of the model or the dataset, and (ii) evaluating these different configurations requires extra time and compute resources. Looking closer at these challenges, however, reveals trivial yet interesting characteristics of ablation studies. When talking about the existence of redundant copies of code for ablation trials, we can observe that these copies are almost identical, except for the part related to the specific ablation trial. Moreover, executing a set of ablation trials is an embarrassingly parallel task.

In this work, we exploit the above characteristics to design and develop a framework to overcome the aforementioned challenges. Our framework, AutoAblation, is based on the concept of the *distribution oblivious training function* [16], in which we decouple *model creation* and *dataset creation* functions from the training function. This decoupling allows us to (i) eliminate the need for maintaining redundant copies of code for ablation trials, and (ii) provide distribution transparency for ML developers so the code that is developed for execution of ablation trials on a single host can easily be executed in parallel on a cluster of machines. This practice has recently enjoyed increased adoption in the community, as can be seen in the programming model of libraries such as PyTorch Lightning [8] and Keras Tuner [19].

With AutoAblation, we introduce a novel way to define and parallelize the execution of ablation studies for (i) DL model architectures, and (ii) training datasets. By decoupling model creation and dataset creation from the training function, we have come up with a simple and declarative Application Programming Interface (API) that eliminates the need for maintaining redundant copies of code for ablation studies. Furthermore, our framework enables parallel execution of ablation trials without requiring the developers to modify their code, which leads to shorter study times and better resource utilization. To the best of our knowledge, this is the first framework that provides support for the specification and parallel execution of ablation studies for DL. We demonstrate the usability and scalability of AutoAblation through three common scenarios in which ablation studies may be performed.

2 Preliminaries

In this section we provide a formal definition of ablation studies in DL, and describe the parallel execution of trials.

2.1 A Formal Definition of Ablation Study

Given a training dataset D and a model M, in training of the model, we aim to optimize its parameters with regards to an objective function (e.g., Mean Squared Error or Binary Cross-Entropy) using an iterative optimization algorithm (e.g., Stochastic Gradient Descent). In practice, developing a performant DL model requires many design decisions and trying out several *configurations* C. The goal of an ablation study is to investigate the relative contribution of each of these configurations to the performance of the model. A configuration can be a *dataset configuration* C_D or a *model configuration* C_M.

The dataset configuration describes what features of a given dataset we need to exclude for training. For a dataset X with n features, $C_D(X, \{x\})$ indicates that the features in $\{x\}$ should be excluded during training the models. For example, $C_D(X, \{x_1, x_3, x_4\})$ means to skip features 1, 3, and 4, and use the rest of the features to train the models. Similarly, the model configuration illustrates the architecture of a model. To be more precise, for a given model M with k components, $C_M(M, \{m\})$ means to exclude the listed components in $\{m\}$ during training M. A component can be a layer, a set of neurons, a filter, and so on. For example, if M is a Convolutional Neural Network (CNN) with two convolution layers (c_1 and c_2), one pooling layer in between (p_1), and one dense layer at the end (d_1), then $C_M(M, \{c_1, p_1\})$ means to remove the first convolution layer and the pooling layer from M and train the model with the rest of component, i.e., $\{c_2, d_1\}$.

We define a *study* S as a set of either dataset configurations, $S_D = \{C_D\}$, or model configurations, $S_M = \{C_M\}$. For example, $S_M = \{C_{M_1}, C_{M_2}, \cdots, C_{M_z}\}$ means to train the model M with z different model configurations, such that in training with configuration C_{M_i}, we only consider the components of the model M, which are not listed in that configuration. We call the execution of an study as an *experiment*, which consists of several *trials*, where each trial corresponds to a configuration. Given the set of components that are to be ablated in an ablation experiment, an *ablation policy* specifies the trials that constitute the experiment. An *ablator*, in turn, is an implementation of an ablation policy that materializes the trials of the experiment. A simple policy could be to remove (exclude) one component per each trial. This is perhaps the most common form of performing ablation studies, and we refer to it as *Leave-One-Component-Out* (LOCO) ablation.

2.2 Parallel Execution of Trials

Over the last few years, several ML and DL frameworks and libraries have been introduced, such as TensorFlow [1], PyTorch [20], and Keras [5]. To make model ablation possible, the underlying DL framework should provide ways for exporting configuration representations of the models, and ways to distinguish different components. All of the above mentioned frameworks fulfill this requirement, e.g., Keras enables developers to set the `name` parameter for layers of a DNN, and export the configuration representation of a model in various formats, such as JSON.

The above mentioned frameworks, however, lack support for parallel and distributed execution of DL experiments consisting of independent trials. For example, TensorFlow and PyTorch provide distributed training capabilities for single models, but practitioners are often left building their own solutions to parallelize their experiments. Therefore, efforts have been made either to develop new frameworks for distributing DL workloads (e.g., Ray [18]), or to use existing distributed processing engines for DL workloads.

Among big data processing frameworks, Apache Spark [29] has been the target of several such efforts [4], as it has become an industry standard for data processing and engineering tasks. TensorFlowOnSpark [28] runs distributed training of a single model with TensorFlow within a Spark job, where each task within this job will serve as a worker process. However, mapping each trial in an experiment to a Spark task results in poor resource utilization, as running iterative jobs on Spark follows the Bulk Synchronous Parallel (BSP) execution model. Stages in Spark introduce task synchronization barriers, and for jobs to proceed to a new stage, all tasks (trials) from the previous stage have to be completed. In case of ablation studies, some trials may take significantly longer time to train due to their configurations, i.e., their specific model architecture, or the dataset subset. Hence, asynchronous execution of trials on an Apache Spark cluster would be highly desirable.

Maggy [15] is a framework for asynchronous execution of trials on Apache Spark clusters. Maggy launches a single Spark job for the whole experiment, and on each Spark executor, one long running task will be run to execute the trials. Once the evaluation of a trial is finished (or stopped) on an executor, the same task on the executor will be reinitialized with a new trial configuration. A `Controller` thread, running on the driver side, is responsible for generating new trial configurations. Depending on the nature of the experiment, the `Controller` can include an *optimizer* (for hyperparameter tuning experiments), or an *ablator* (for ablation studies). Currently, Maggy supports asynchronous, parallel execution of hyperparameter tuning experiments [15], and with AutoAblation we extend it to also support ablation study experiments.

Through a communication model based on non-blocking Remote Procedure Calls (RPC), once the job is launched (or an executor finishes an assigned trial), the executors can communicate with the driver and poll the controller for new trials independent of other executors. This removes the barrier (synchronization step) imposed by the BSP execution model, leading to increased resource utilization as well as reduced total run-time of experiments.

3 AutoAblation

Currently, AutoAblation supports model ablation and feature ablation of DNNs. Model ablation is possible in form of individual layers, groups of layers, and modules (e.g., an Inception module), and feature ablation is possible in form of individual features or groups of features. To address the two challenges of (i) redundant code maintenance, and (ii) efficient parallel execution of ablation trials, we exploit the fact that the training logic remains largely unchanged between different trials. When we want to investigate the contribution of different components of a DL model to its overall performance, we have to construct different variants of the model architecture and apply the same training logic on these variants, using the same training data. Similarly, if we are interested in the importance of each feature of our training dataset, we use different combinations of the features to train the same model, using the same training logic.

Following the above observations, the programming model of AutoAblation is based on the decoupling of the model creation and dataset creation from the training logic. In other words, instead of having model creation, dataset creation, and training logic in a single block of code, the user wraps the training code in a function that is parameterized by dataset creation and model creation functions. This decoupling and parameterization allows the framework to automatically generate and replace parts of the logic that are specific to each trial. Performing an ablation experiment in AutoAblation consists of three steps[1]: (i) defining the training components (including model creation and dataset creation), (ii) defining the ablation study, and (iii) executing the ablation trials in parallel. We will explain this workflow, as shown in Figure 2, in the following subsections.

[1]See https://maggy.readthedocs.io/ for the API documentation.

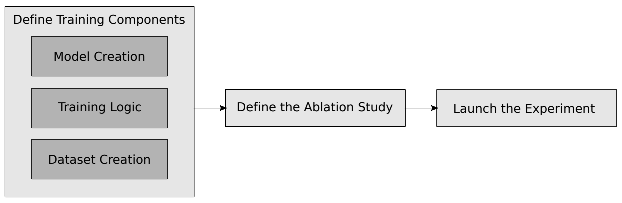


Figure 2. Workflow of an ablation experiment.

3.1 Defining the Training Components

The first step is to define the actual training loop. This step is always part of the ML process, irrespective of whether an ablation study will be performed or not. The important thing here, however, is that the user has to decouple the model creation and the dataset creation from the training function. In most cases, this is equivalent to moving the code blocks responsible for the model creation and the dataset creation to their own functions, e.g., `create_model()` and `create_dataset()`, and passing them as arguments to the training function. A skeleton code for the first step is shown in Listing 1.

Model Creation. Here, the user has to wrap the model creation code in a Python function that we refer to as the *base model* function, which receives trial-specific parameters (e.g., layer identifiers) and returns a trainable model that can be used in the training function.

Dataset Creation. Similar to model creation, the process of creating the train/test/validation sets that will be used in the train/test/validation loops should be wrapped in its own function. The user can implement their own function for creating these sets or use default dataset creation functions shipped with AutoAblation.

Training Function. The training function is the actual pure Python code block that will be executed either on a single host or in parallel on a cluster of workers, and contains the code for training a DL model using a training dataset. In a typical implementation of a DL application, the whole process of preparing the train/test/validation sets, model architecture definition, and model training is implemented in a monolithic style; but in our programming model, as the user has already implemented the model creation and the dataset creation functions in the previous sub-steps, the model function and the dataset function are passed as arguments to the training function, and will instantiate the model and the dataset(s) once called.

3.2 Defining the Ablation Study

The next step is to define the ablation study by specifying the model configuration and dataset configuration (S_D and S_M, as defined in Section 2.1). To this end, the user has to create an `AblationStudy` instance and initialize it with the default model creation and dataset creation functions defined in the previous step. After this, the user should specify which configurations they want to *include* in the study. Currently, AutoAblation API provides methods for defining configurations for dataset features, model layers, layer groups, modules, and custom models. Example usage of the API for defining an ablation study is shown in Listing 2.

```python
# define the model creation logic
def base_model(trial_params):
    # create the model ...
    return model

# define the dataset creation logic
def base_dataset(trial_params):
    # create the dataset ...
    return dataset

# define the training logic, parametrized by the model and dataset
def train(model_func, dataset_func):
    model = model_func()
    data = dataset_func()
    metric = model.fit(data)
    return metric
```

Listing 1. Defining the training components.

```python
# define the ablation study
study = AblationStudy()

study.model.set_base_model_generator(base_model)
study.set_dataset_generator(base_dataset)

study.features.include('feature_name')
study.model.layers.include('layer_name')
study.model.add_module('module_name')

# launch the experiment
experiment.launch(train, study)
```

Listing 2. Defining the ablation study.

3.3 Launching the Experiment

The final step is to invoke an API call that mainly receives the training function and the study specification, and initiates the execution of the trials through Maggy, either sequentially on a single host or in parallel on a cluster of nodes.

4 Implementation

AutoAblation runs on top of Maggy, an open-source Python-based framework for asynchronous execution of ML trials on top of Apache Spark. The experiment is launched as a Spark application that generates the trials of the experiment, and Maggy distributes the trials on the set of available worker nodes (executors). Below, we explain how AutoAblation generates trials based on the ablation study specification defined in the second step of the workflow in Section 3.

4.1 Implementing the LOCO Ablator

As discussed in Section 2.1, an ablator is an implementation of an ablation policy. In AutoAblation, ablators are implemented as Python classes. The `Controller` thread in the

Spark job creates an instance of an ablator class and uses it to generate `Trial` objects that contain the model creation and dataset creation functions specific to each trial. To execute a trial, an executor requests a new trial configuration from the `Controller`. If there is a trial to be evaluated, the executor will be sent a `Trial` object. The executor then de-serializes the object and unpacks its contents, and then passes them as arguments to the training function, and executes the training function.

The LOCO ablator uses the dataset creation and model creation functions to generate `Trial` objects that are then shipped to the executors as they request new trial configurations. Given an `AblationStudy` instance that contains different configurations, in order to create customized models and datasets for each trial, an ablator must modify the base model or dataset by removing these components. LOCO does this through parsing and modifying "configuration representations" of the components. For datasets, this is equivalent to the dataset schema (which can come in different formats). For models, many DL frameworks provide ways for saving or exporting model configurations, e.g., through JSON files (as in Keras) or serializable dictionaries (as in PyTorch). Hence, an ablator essentially implements the process of parsing and modifying these configuration representations and generating trials according to an ablation policy.

Given the schema of the base dataset, the LOCO ablator modifies the base schema to create a new schema for each feature ablation trial, and generates its corresponding `create_dataset` function. To generate `create_model` functions specific to each trial, the LOCO ablator uses the base model function (as described in Section 3.1) to export the configuration representation of the base model, and then parses it to find and select model components defined in the `AblationStudy` instance of the experiment. It then modifies the configuration representations and generates new `create_model` functions for each trial.

Input or output shape changes that may result from removal of components are either handled by the underlying framework (e.g., when removing layers of a model developed with Keras Sequential API), or require explicit handling in the implementation of the ablator (e.g., by using a randomly initialized tensor as the input of one forward pass of the modified model, to infer the correct shapes). However, if a trial cannot be automatically generated, the user still has the option to create a custom trial with their own model creation and dataset creation functions, and add it to the experiment. Finally, the LOCO ablator creates the corresponding `Trial` objects, and populates the buffer of trials that the executors can poll as the experiment is launched.

4.2 Parallel Execution of Trials

To execute different trials of an experiment, each Spark executor needs to have the training function that is parametrized by the `create_model` and `create_dataset` function objects. The training function is supposed to remain the same through all trials, so it will be sent to the executors as the experiment is launched. The two parameters of the training function, will be provided through the `Trial` objects created by the LOCO ablator. The executors will then register with the Maggy driver, and start polling the server for these objects. Depending on the ablation policy, a number of initial trials will be generated on the driver side; in the case of LOCO, since the number of trials can be determined from the components included in the ablation study, the ablator will generate all trials and put them in a buffer, which will be queried by the `Controller` every time an executor requests a new trial configuration. It should be noted that the Spark job is started and managed by Maggy, and the start-up only takes a few seconds, which is negligible compared to the actual time it takes to train the model variants.

5 Evaluation

In this Section, we demonstrate three common scenarios, in which ablation studies can be performed, and show how we can define and execute such studies with AutoAblation[2]. Below, we first evaluate the performance of AutoAblation in two different experiments: (i) feature ablation and (ii) model ablation, and then we show how it performs in various levels of parallelization.

EXP1: Feature Ablation of the Titanic Dataset. In this experiment, we perform feature ablation on a customized version of the Titanic dataset[3]. There are six features in the dataset in addition to the label, so we will have seven trials (including one base trial that contains all the features). The model we use is a simple Keras Sequential model with two hidden `Dense` layers. We keep 20% of the data as the test set and train on the rest for 10 epochs. Listing 3 shows the code required to define this experiment.

```
from maggy.ablation import AblationStudy
study = AblationStudy('titanic_train_dataset', label_name='survived')
list_of_features = ['pclass', 'fare', 'sibsp', 'sex', 'parch', 'age']
study.features.include(list_of_features)
```

Listing 3. Defining the feature ablation experiment.

After repeating the experiment five times, we can rank the features in terms of their average effect on the test accuracy, as shown in Table 1. For example, we observe that training the model with all the features (`None`) results in the worst test accuracy, while removing the `fare` feature from the training dataset leads to the best performance.

EXP2: Model Ablation of a Keras Sequential Model. In this experiment, we train a CNN to classify handwritten digits of the MNIST dataset [12]. The network has two `Conv2D`

[2]The reproducible experiments: https://github.com/ssheikholeslami/ablation-paper-experiments

[3]https://www.kaggle.com/c/titanic/data

Table 1. Average accuracy on the test set resulting from excluding each feature from the training set.

Excluded Feature	Test Accuracy
None (base trial)	0.583
pclass	0.596
sex	0.609
sibsp	0.616
age	0.667
parch	0.672
fare	0.695

layers, followed by one `MaxPooling2D` layer, one `Dropout` layer, a `Flatten` layer, one `Dense` layer, another `Dropout` layer, and one `Dense` output layer. Our target is to investigate the relative contribution of the second `Conv2D` layer, the `Dense` layer, and the first and second `Dropout` layers to the performance of the model. The study can be defined using the code shown in Listing 4. After repeating the experiment five times, we can rank the selected layers in terms of their average effect on the test accuracy, as shown in Table 2. We can see that removing the second `Conv2D` layer has the worst effect on the test accuracy, while removing the `Dropout` layers results in a better performance than the performance of the base model.

```
from maggy.ablation import AblationStudy
study = AblationStudy("mnist", 1, "number",)
study.model.layers.include('second_conv',
        'first_dropout', 'dense_layer', 'second_dropout')
```

Listing 4. Defining the CNN model ablation experiment.

Table 2. Average accuracy on the test set resulting from excluding layers of interest from the base model.

Excluded Layer	Test Accuracy
second_conv	0.913
dense_layer	0.954
None (base trial)	0.969
second_dropout	0.982
first_dropout	0.988

EXP3: Model Ablation of Inception-v3. With this experiment, we demonstrate the near-linear scalability achieved by parallel execution of ablation trials with AutoAblation. We perform an ablation study on seven modules of the Inception-v3 network [25] in a transfer learning task on a subset of the TenGeoPSAR dataset [26]. This subset contains 5000 Synthetic Aperture Radar (SAR) images. We split the dataset into train (3200 images), validation (800 images), and test (1000 images) sets. The images are labeled with one of 10 classes, each representing a geophysical phenomena.

We load the network using Keras Applications API with pre-trained ImageNet [6] weights, and replace its output layer to suit our 10-class classification task. The Inception-v3 network consists of 11 blocks also known as "inception modules", and we are interested to know how each of the first seven modules affect the performance of the network (measured by the accuracy on the test set). Since this is a predefined network, we first compile it to find out about the names of the layers, and identify the entrance and end point of each module either by plotting the architecture or observing the `model.summary()` output information in Keras. Once we identify the layers, defining the ablation study can be done with the code shown in Listing 5.

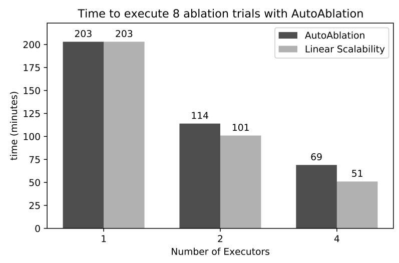


Figure 3. AutoAblation provides near-linear scalability by parallelizing the execution of ablation trials.

```
from maggy.ablation import AblationStudy
study = AblationStudy("TenGeoPSARwv", 1, "type",)
study.model.add_module('max_pooling2d_1', 'mixed0')
study.model.add_module('mixed0', 'mixed1')
study.model.add_module('mixed1', 'mixed2')
...
study.model.add_module('mixed5', 'mixed6')
```

Listing 5. Defining the Inception-v3 module ablation experiment.

Each trial consists of 40 epochs of training, and we run the experiment in three settings: (i) a single executor (sequential, no parallelization), (ii) two executors, and (iii) four executors. The total run-time for each of these settings is reported in Figure 3. We take the run-time of the sequential run as a baseline to approximate linear scalability; however, we should keep in mind that the ablation trials differ in their run-time since their model architectures are different from one another. We can conclude from Figure 3 that AutoAblation provides near-linear scalability by parallelizing the execution of ablation trials.

6 Related Work

Recently there have been many efforts to build frameworks, libraries, and tools to inquire insights regarding the performance of DL models or the effect of different dataset configurations in their performance. Many of such efforts address the problem of Interpretability and Explainability of ML/DL models[4]. Libraries such as LIME [21], SHAP [14], and TensorFlow's What-If Tool [27] provide extensive tools and visualizations for explaining the behaviour and outputs of ML/DL models through *post-hoc analysis*. DeepBase [23] is a system for deep neural inspection that provides a declarative

[4]A list of related open-source projects can be found in: https://github.com/EthicalML/awesome-production-machine-learning

API for defining hypothesis functions and then evaluates those hypotheses over a sequence of inputs. DeepBase is similar to AutoAblation as it shares a design requirement to reduce the amount of effort for performing model inspections, but with AutoAblation the same code can be used for hyperparameter tuning, distributed training, and other types of DL experiments [16]. LOFO-Importance [7] is a library that provides Leave-One-Feature-Out importance for datasets used to train models, by excluding one feature out of the training set at a time, and retraining the model on that subset. However, it does not provide support for model ablation experiments.

7 Conclusion and Future Work

In this paper, we introduced AutoAblation, a new framework for the design and parallel execution of ablation studies of deep learning models. We formulated an ablation study as an experiment that consists of several trials, where each trial represents a specific model architecture or dataset schema. We also presented a new programming model for designing an experiment that is based on the decoupling of model creation and dataset creation from the training function. We introduced the concept of the ablation policy that specifies what should be the trials that make up an ablation experiment, implemented in form of an ablator. Moreover, we showed how we leverage parallel execution of trials to speed up the total study time and increase resource utilization, through our Python-based execution framework called Maggy. Through the experiments, we showed that AutoAblation provides near-linear scalability. Our next step would be to develop a generalized approach for handling shape mismatch issues, and to support automatic generation of more complex ablation trials and policies, e.g., cases in which removal of a layer requires other changes in the components of a model. As AutoAblation gets picked up by more users, we will use their feedback to provide support for more common ablation scenarios.

Acknowledgments

This work is supported by the ExtremeEarth[5] project funded by European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 825258. The authors would like to thank Farzad Nozarian and Desta Hagos for their detailed feedback on the draft of this paper.

References

[1] M. Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[2] D. Berthelot et al. 2019. MixMatch: A Holistic Approach to Semi-Supervised Learning. *arXiv preprint arXiv:1905.02249* (2019).

[3] N. Carlson et al. 2009. *Psychology: the Science of Behavior.* Pearson.

[4] B. Chambers and M. Zaharia. 2018. *Spark: The Definitive Guide: Big Data Processing Made Simple.* O'Reilly Media, Inc.

[5] F. Chollet et al. 2015. Keras.

[6] J. Deng et al. 2009. Imagenet: A Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition.* IEEE, 248–255.

[7] A. Erdem et al. 2019. Leave One Feature Out Importance. https://github.com/aerdem4/lofo-importance.

[8] W. A. Falcon et al. 2019. PyTorch Lightning. *GitHub. https://github.com/PyTorchLightning/pytorch-lightning* 3 (2019).

[9] R. Girshick et al. 2014. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 580–587.

[10] M. Hessel et al. 2018. Rainbow: Combining improvements in deep reinforcement learning. In *33 AAAI Conference on Artificial Intelligence.*

[11] E. Horvitz et al. 2003. Learning and reasoning about interruption. In *Proceedings of the 5th International Conference on Multimodal Interfaces.* ACM, 20–27.

[12] Y. LeCun. 1998. The MNIST Database of Handwritten Digits. http://yann.lecun.com/exdb/mnist/.

[13] Z. C. Lipton and J. Steinhardt. 2018. Troubling trends in machine learning scholarship. *arXiv preprint arXiv:1807.03341* (2018).

[14] S. M. Lundberg and SI. Lee. 2017. A Unified Approach to Interpreting Model Predictions. *Advances in Neural Information Processing Systems* 30 (2017), 4765–4774.

[15] M. Meister et al. 2020. Maggy: Scalable Asynchronous Parallel Hyperparameter Search. In *Workshop on Distributed Machine Learning.* 28–33.

[16] M. Meister et al. 2020. Towards Distribution Transparency for Supervised ML With Oblivious Training Functions. In *Workshop on MLOps Systems.*

[17] R. Meyes et al. 2019. Ablation Studies in Artificial Neural Networks. *arXiv preprint arXiv:1901.08644* (2019).

[18] P. Moritz et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* 561–577.

[19] T. O'Malley et al. 2019. Keras Tuner. https://github.com/keras-team/keras-tuner.

[20] A. Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32 (2019), 8026–8037.

[21] M. T. Ribeiro et al. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* 1135–1144.

[22] M. Richardson et al. 2006. Beyond PageRank: Machine Learning for Static Ranking. In *Proceedings of the 15th International Conference on World Wide Web.* ACM, 707–715.

[23] T. Sellam et al. 2019. DeepBase: Deep Inspection of Neural Networks. In *Proceedings of the 2019 International Conference on Management of Data.* 1117–1134.

[24] S. Sheikholeslami. 2019. *Ablation Programming for Machine Learning.* Master's thesis.

[25] C. Szegedy et al. 2016. Rethinking the Inception Architecture for Computer Vision. In *IEEE Conference on Computer Vision and Pattern Recognition.* 2818–2826.

[26] C. Wang et al. 2019. A labelled ocean SAR imagery dataset of ten geophysical phenomena from Sentinel-1 wave mode. *Geoscience Data Journal* 6, 2 (2019), 105–115.

[27] J. Wexler et al. 2019. The What-If Tool: Interactive Probing of Machine Learning Models. *arXiv preprint arXiv:1907.04135* (2019).

[28] L. Yang et al. 2017. Open Sourcing TensorFlowOnSpark: Distributed Deep Learning on Big-Data Clusters.

[29] M. Zaharia et al. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* 10, 10-10 (2010), 95.

[5]ExtremeEarth project website: http://earthanalytics.eu .