Noname manuscript No.
(will be inserted by the editor)

CalcGraph: Taming the high costs of deep learning using models

Joe Lorentz · Thomas Hartmann · Assaad Moawad · Francois Fouquet · Djamila Aouada · Yves Le Traon

Received: date / Accepted: date

Abstract Models based on differential programming, like deep neural networks, are well established in research and able to outperform manually coded counterparts in many applications. Today, there is a rising interest to introduce this flexible modeling to solve real-world problems. A major challenge when moving from research to application are the strict constraints on computational resources *(memory and time)*. It is difficult to determine and contain the resource requirements of differential models, especially during the early training and hyperparameter exploration stages. In this article, we address this challenge by introducing CalcGraph, a model abstraction of differentiable programming layers. CalcGraph allows to model the computational resources that should be used and then CalcGraph's model interpreter can automatically schedule the execution respecting the specifications made. We

The research leading to this publication is supported by the National Research Fund Luxembourg (grant 14297122)

Joe Lorentz
DataThings S.A. & University of Luxembourg, Luxembourg, Luxembourg E-mail: joe.lorentz@datathings.com

Thomas Hartmann
DataThings S.A., Luxembourg, Luxembourg

Assaad Moawad
DataThings S.A., Luxembourg, Luxembourg

Francois Fouquet
DataThings S.A., Luxembourg, Luxembourg

Djamila Aouada
University of Luxembourg, Luxembourg, Luxembourg

Yves Le Traon
University of Luxembourg, Luxembourg, Luxembourg

propose a novel way to efficiently switch models from storage to preallocated memory zones and vice versa to maximize the number of model executions given the available resources. We demonstrate the efficiency of our approach by showing that it consumes less resources than state-of-the-art frameworks like TensorFlow and PyTorch for single model and multi-model execution.

Keywords differentiable programming · computational graph model · edge AI

1 Introduction

Artificial intelligence (AI) and machine learning (ML) are currently receiving much attention. In the past few years, this technology has left the realm of pure research behind. Advanced machine learning models, especially deep neural networks, have been successfully applied to many practical domains, *e.g.,* machine translation [57], image recognition [28], self-driving cars [17], the medical sector [37], and automation [40]. Therefore, more and more companies understandably start to explore how AI and ML can be an asset to their businesses. However, transitioning AI from research to real world applications, comes with some major challenges. ML models, especially deep neural networks, can easily exceed the available computational resources, when they are not implemented efficiently. It is pivotal that implementations use the limited resources efficiently to enable as many executions of complex models within a given time-frame as possible. The execution speed of ML models is especially critical for applications that are supposed to take decisions in real-time. Examples for this are self-driving cars [17] or quality assurance systems on fast moving production lines [56]. Both examples require classifying multiple frames per second. In addition to time, a second major resource is the available memory. Memory-effecient ML implementations allow more complex models to be used on the same device, potentially increasing the robustness of AI-driven solutions. In addition, for many companies it can be crucial to allow multiple users or applications on the same device while limiting the memory consumption of individual entities. In such situations, it is necessary to predict the memory requirements of the process before starting it and containing the consumption throughout its execution. Time and memory constraints are especially relevant if cloud computing is not wanted or applicable and solutions should be integrated on the edge [40]. This is often the case in the context of Industry 4.0 and Internet of Things. Here, cloud solutions can be unsuitable due to security concerns, connection reliability or latency. Currently, edge AI research focuses on fast efficient inference to enable AI for real-time applications. However, as the software and hardware components become more efficient, practitioners might also want to train models on the same devices and use the innate flexibility of ML models to adapt to changing situations [47].

In this article, we introduce CalcGraph, with the goal to reduce and contain the high costs of ML models. CalcGraph is a model abstraction of differentiable programming layers that underlie deep learning algorithms. It has already been shown that model-driven engineering and machine learning can complement one another in many ways [16, 26, 27]. We leverage models (CalcGraphs), similar to how some compilers use abstract syntax trees (ASTs), to drive and optimize the compilation process [65]. More specifically, our contribution comprises the following three points:

1. Our model compiler generates optimized branchless byte code that can be executed as a fixed sequence of operations. Most importantly, this allows to compute the *required main- and GPU memory size* for a CalcGraph

model and a given batch size—or an optimal batch size for a given memory size—*at compile time.*

2. We minimize expensive memory allocations by preallocating the entire needed memory (computed by the model compiler) and then reusing it by switching models from storage to the preallocated memory and vice versa. This avoids expensive *malloc* and *free* operations during runtime and allows us to explore different ML models with constant memory. We refer to this as *model switching.*
3. Knowing the memory requirements at compile time enables us to further *optimize the usage of the available resources* by scheduling as many models as possible for *parallel execution* and splitting the executions among the available resources.

Determining the optimal ML solution for a given problem requires to explore many models. Hence, reducing the cost of model switches and executing as many models as possible in parallel—on the given resources—is key to reduce the high costs of differential programming in terms of execution time and memory consumption. CalcGraph allows to model the available resources *(memory, time)* and then CalcGraph's model interpreter can automatically schedule the execution respecting this specification. Rather than focusing on the exploration level, we focus with CalcGraph on providing an efficient foundation on which model exploration can be built on top. We validate the efficiency of our approach by showing that it can outperform model exploration built on top of state-of-the-art frameworks, like TensorFlow and PyTorch, in terms of execution time as well as memory consumption while reaching the same accuracy.

The remainder of this paper is as follows. In Section 2 we first present the background for this article: differentiable programming, foundations of deep learning and the challenges towards real world application. Then, in Section 3 we present our proposed CalcGraph model in detail, the challenges of applying deep learning to real world problems and how we approach them with CalcGraph. In Section 4 we evaluate our approach, discuss future work in Section 5, related work in Section 6, and conclude in Section 7.

2 Background

In this section, we first introduce differentiable programming, as our proposed CalcGraph is an abstraction over differentiable programming layers. We then explain how deep learning and differentiable programming are related. Lastly, we introduce the challenges of deep learning for real-world applications and describe how CalcGraph can be used to tackle them.

2.1 Differentiable programming

The term differentiable programming (also referred to as *DiffProg* or ∂P) has lately been popularized by Yann LeCun, who describes it as "a modern rebranding of deep learning, similar to how deep learning was a modern rebranding of neural networks with more than two layers" [1]. As we will explain in the next section, differentiable programming is a programming model in which arbitrary programs can be differentiated, usually via *automatic differentiation* [33, 62]. Differentiation allows to systematically optimize the parameters of a program, *i.e.,* to find the "good" values of the parameters, using *gradient-based* optimization (often gradient descent). In other words, differentiable programming allows to use deep neural networks as building blocks in and for our programs. Andrej Karpathy introduced the term *Software 2.0* [7] to indicate a shift in the way software can be written using deep neural networks. Interestingly, this idea is close to ideas of model-driven engineering, where a code generator can automatically generate the code for a problem specified by a model – just in the case of Karpathy's vision of Software 2.0 the generator is automatically derived (or learned) via optimization instead of manually programmed. In this direction, in a previous work [25], we discussed how machine learning can be seamlessly integrated into modeling languages. Although differentiable programming is more generic and has found use in a wide variety of areas, particularly scientific computing, it is nowadays prominently used in machine learning, or more specifically deep learning. This is also the area where we focus our work on.

2.1.1 Automatic differentiation

Automatic differentiation (AD) refers to a set of techniques to compute derivatives of functions by applying the chain rule in recursion to accumulate the final values from intermediate evaluations [9]. The simpler intermediate steps and the accumulation of AD are implemented in code, therefore it is also referred to as algorithmic differentiation or computational differentiation [43]. Computing derivatives in machine learning with AD has some major advantages over the alternative numerical and symbolic differentiation. The approximations used in numerical differentiation are easy to implement but introduce approximation errors and have a computational complexity of $O(n)$ with n the number of gradient dimensions [9]. Symbolic differentiation is exact, however, expressions can grow exponentially, which makes it difficult to implement the approach efficiently. AD heavily uses recursion and stores intermediate results in memory to evaluate function derivatives to working precision while only executing the underlying function code a constant number of times. There are two main approaches of automatic differentiation: forward-mode AD and reverse-mode AD. We will use a simple example to illustrate both modes. Let us consider the function $f(x_1, x_2)$ and define:

$$\begin{aligned} f(x_1, x_2) &= x_1 x_2 + exp(x_1) - cos(x_2) \\ \omega_{00} &= x_1 \\ \omega_{01} &= x_2 \\ \omega_1 &= \omega_{00}\omega_{01} \\ \omega_2 &= exp(\omega_{00}) \\ \omega_3 &= -cos(\omega_{01}) \\ \omega_4 &= \omega_1 + \omega_2 \\ \omega_5 &= \omega_4 + \omega_3 \end{aligned} \tag{1}$$

For both modes the computations of $f(x_1, x_2)$ at a the evaluation point needs to be performed to get the respective values of the ω_i. Forward AD starts with partial derivatives at inputs and ends by computing partial derivatives at outputs. In our example, forward AD requires two executions to compute the derivatives with respect to the two independent variables. The derivative with respect to x_1 (ω_{00}) is computed with the following operations:

$$\begin{aligned} \dot\omega_{00} &= 1(init) \\ \dot\omega_{01} &= 0(init) \\ \dot\omega_1 &= \dot\omega_{00}\omega_{01} + \omega_{00}\dot\omega_{01} = \omega_{01} \\ \dot\omega_2 &= \dot\omega_{00} exp(\omega_{00}) = exp(\omega_{00}) \\ \dot\omega_3 &= \dot\omega_{01} sin(\omega_{01}) = 0 \\ \dot\omega_4 &= \dot\omega_1 + \dot\omega_2 = \omega_{01} + exp(\omega_{00}) \\ \dot\omega_5 &= \dot\omega_4 + \dot\omega_3 = \omega_{01} + exp(\omega_{00}) + 0 \end{aligned} \tag{2}$$

Reverse AD operates in the opposite direction, starting with partial derivatives at the output and backpropagating via so-called adjoints $\bar\omega_i = \frac{\vartheta y_i}{\vartheta \omega_i}$. Using the same example as before, the computation would be performed as follows:

$$\begin{aligned} \bar\omega_5 &= 1(init) \\ \bar\omega_4 &= \bar\omega_5 * 1 = 1 \\ \bar\omega_3 &= \bar\omega_5 * 1 = 1 \\ \bar\omega_1 &= \bar\omega_4 * 1 = 1 \\ \bar\omega_2 &= \bar\omega_4 * 1 = 1 \\ \bar\omega_{00}^a &= \bar\omega_1 \omega_{01} = \omega_{01} \\ \bar\omega_{00}^b &= \bar\omega_2 exp(\omega_{00}) = exp(\omega_{00}) \\ \bar\omega_{00} &= \bar\omega_{00}^a + \bar\omega_{00}^b = \omega_{01} + exp(\omega_{00}) \\ \bar\omega_{01}^a &= \bar\omega_1 \omega_{00} = \omega_{00} \\ \bar\omega_{01}^b &= \bar\omega_3 sin(\omega_{01}) = sin(\omega_{01}) \\ \bar\omega_{01} &= \bar\omega_{01}^a + \bar\omega_{01}^b = \omega_{00} + sin(\omega_{01}) \end{aligned} \tag{3}$$

Reverse AD computes the partial derivatives with respect to an arbitrary number of inputs at one go. In general reverse AD is faster than forward AD when the number of inputs is large in comparison to the number of outputs [43]. This makes reverse mode advantageous for machine learning where the gradient with respect to thousands or even millions of parameters need to be computed. In fact, **reverse-mode AD is a more general form of the backpropagation algorithm** used in multilayer perceptrons, *i.e.,* in deep learning using modern neural networks [9]

2.1.2 Approaches to automatic differentiation

There are different approaches when it comes to implement automatic differentiation. One is to use *domain-specific languages (DSLs).* Automatic differentiation can be implemented as an external or internal DSL. External DSLs are their own languages with their own syntax and semantics, whereas internal DSLs are embedded into a host language and can for example be imported via a library. While embedded DSLs have without doubt their advantages, they come also with limitations: they are often restricted to specific types and APIs, usually involve some boilerplate to map to their host language, are restricted to the meta-programming capabilities of their host language, and may not provide good diagnostic support. Another approach is the use of *source code transformation tools*, *i.e.,* code generators [25]. This means the tool analyzes the input code and generates the output code that computes the derivatives. Such tools are essentially static compilers. This allows to perform static analyses on the input code and to generate optimized output code. For example, it can be determined if some variables do not need a derivate or if some intermediate variables can be removed. A disadvantage is that such tools are often not well integrated into the programming environment and therefore creating an extra step in the workflow. *First-class language support* is another approach to differentiable programming. The idea is to combine the advantages of source code transformation tools and a seamless integration into host languages. Currently most programming languages that support differentiable programming are research systems, *e.g.,* [31,46], but a proposal to back differentiable programming into a mainstream programming language, namely Swift, is currently under work [63].

With CalcGraph, we essentially follow the source code transformation approach. CalcGraph models are compiled into optimized byte code, which is then executed via an interpreter. However, to mitigate the mentioned disadvantages of source code transformation approaches, we develop our own external DSL for CalcGraph. In addition to our own language, we also support ONNX models. Since our language is not of further importance in the context of this article, we consider it out of scope.

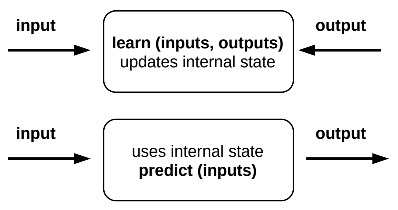

Fig. 1: A neural network as a black box

2.2 Deep learning

Over the past few years machine learning has been incredibly successful, especially deep neural networks. A deep neural network is an artificial neural network with multiple layers between input and output [10]. Accordingly, the term deep learning is used when deep neural networks are involved in the learning algorithms. In this section, we present an overview of a neural network and show how differentiable programming and deep learning are linked.

2.2.1 Overview of a neural network

At a high-level, a supervised neural network can be presented as a black box with two methods, *learn* and *predict* as shown in Figure 1. The learn function uses the inputs and outputs to update the internal state of the neural network and create an internal knowledge. The predict function is used to infer the outputs from the given inputs.

The first step in learning is to start from somewhere, the initial hypothesis. Like in genetic algorithms and evolution theory, neural networks can start from anywhere. Thus, a random initialization of the weights (internal state) is a common practice. The rational behind is that from wherever we start, through an iterative learning process, we can reach the pseudo-ideal model.

The next step, after initializing the model randomly, is to check its performance. We start from the inputs we have, pass them through the network and calculate the actual output of the model in a straightforward manner. This step is called *forward-propagation*, because the calculation flow is going in the natural forward direction from the inputs, through the neural network, to the outputs. To be able to generalize our model, we define what is called a *loss function*. A loss function is a performance metric on how well the neural network is able to reach its goal of generating outputs as close as possible to the desired values. Simply speaking, the machine learning goal is to minimize the loss function (to reach as close as possible to 0). Now, we can transform our machine learning problem to an optimization problem that aims to minimize this loss function.

We could use any optimization method that modifies the internal weights of neural networks in order to minimize the total loss function, *e.g.,* genetic algorithms, greedy search, or even a simple brute-force search. While this might work if the model is simple and has only very few parameters, however, if we are training the neural network over many inputs (like in image processing), we reach quickly models with millions of weights to optimize. This is where differentiation comes into the game.

2.2.2 Backpropagation and automatic differentiation

Differentiation deals with the derivative of the loss function. In mathematics, the derivative of a function at a certain point, gives the rate or the speed of which this function is changing its values at this point. As explained in Section 2.1.1, derivatives are decomposable, thus can be *back-propagated. Automatic differentiation* technique requires only that each function is provided with the implementation of its derivative.

Since we have the starting point of errors, which is the loss function, and since we know how to calculate the derivative of each function from the composition, we can propagate back the error from the end to the start. If we create a library of differentiable functions or layers where for each function we know how to forward-propagate (by directly applying the function) and how to back-propagate (by calculating the derivative of the function), we can compose arbitrary complex neural networks. Then, we only need to keep track of the function calls during the forward pass and their parameters in order to know the way back to back-propagate the errors using the derivatives of these functions. As we will see later (Section 3), this is what we provide with our proposed CalcGraph. We implement for each operator that can be modeled in the CalcGraph a forward and backward mode.

Using the gradients, the learning process at any layer looks as follows:

– Check the gradient of an element.
– If it is positive, meaning the error increases if we increase the weights, then we should decrease the weight.
– If it is negative, meaning the error decreases if we increase the weights, then we should increase the weight.
– If it is 0, we do nothing, the error is at its minimum. We are at our stable point.

This process is depicted in Figure 2.

Since in real-world problems there are usually a lot of non-linear functions involved, any big change in weights will lead to a chaotic behavior. It is important to keep in mind that the gradient is only local at the point where we are calculating it. Therefore, weights should be only updated in very small steps. Several weight update methods exist, these methods are often called *optimizers.* The loss function could be optimized over the whole dataset, this is called *batch learning.* Another option is to update the weights every n elements of the training data, expecting that the dataset is shuffled randomly. This is called *mini-batch gradient descend.* And if $n = 1$, we call it *full online-learning or stochastic gradient descent*, since we are updating the weights after each single input/output observed. Any optimizer can work with these 3 modes (full

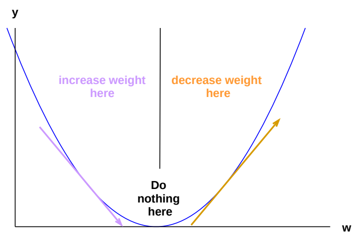

Fig. 2: Arrows represent the gradients at the corresponding points

online/mini-batch/full-batch). Since we update the weights with a small delta step at a time, it will take several iterations in order to learn.

2.3 Challenges of deep leaning in real world applications

Despite the proven effectiveness of Deep Learning in research, moving to real world applications has some major challenges.

2.3.1 Computational resource constraints

Since their introduction, neural networks have become both deeper and wider to improve accuracy [29, 59]. Today it is common in research to use models with several hundreds of layers and millions of parameters. The downside of this evolution are ever rising computational requirements. Huge models like this require equally big amounts of memory to store all the parameters, even more so in case of training where gradients also need to be stored and accumulated during backpropagation. The computational complexity, i.e. the execution time, is also rising, despite efforts to optimize the efficiency on the architectural level [29, 51].

In research labs, this challenge is usually solved by using high-end GPUs or even a multitude of them. This solution is not feasible for many real world applications due to cost or power consumption. This is particularly important if cloud computing is unfeasible or unwanted and the hardware solutions need to be replicated for many edge-instances [40]. For Deep Learning to be successful in real world, a hardware-software combination needs to be defined that can be reproduced in a cost-effecient way. This need has recently been acknowledged by hardware manufacturers, which lead to the introduction of GPUs on small factor computing units like Nvidia's Jetson family [58]. On the software side, a proper management of the available resources is required. It is pivotal that implementations use the limited resources efficiently to enable as many executions of complex models within a given time-frame as possible. The execution speed of ML models is especially critical for applications that are supposed to take decisions in real-time. An example for this are quality assurance systems on fast moving production lines, which require to classify multiple frames per second [56]. Memory-effecient ML implementations allow more complex models to be used on the same device, potentially increasing the robustness of AI-driven solutions. In addition, for bigger companies it can be crucial to allow multiple users or applications on the same device while limiting the memory consumption of individual entities. In such situations, it is necessary to predict the memory requirements of the process before starting it and containing the consumption throughout its execution. Our CalcGraph allows for this by pre-compiling models and allowing users to set strict limits on the allocated memory.

Improving the efficiency of software and hardware components for applications on the edge might even allow for advanced use cases. While today, edge-AI is mostly used for efficient inference, advanced applications could also aim for continuous training cycles directly on the integrated production setup. Aside from forward and backward on neural networks, there might also be additional computations that need to be handled. For example, the input data might not be directly available in a format that is suitable for the differential models. In such cases, some additional code execution to pre-process the data might be needed. Post-processing steps on the model outputs are also commonly needed. Among them, a challenging and ML-specific task is to provide reasoning for the model decisions. Lacking explainability of deep neural networks is direct consequence of the flexibility and autonomous feature learning they provide [49]. Neural networks tend to rely heavily on features which are un-intuitive for human perception [52]. This is a major issue for a number of real world applications where decisions need be to traceable or human-machine-interaction is wanted. Consider for example AI-driven judging for tribunal decisions. Here it is not good enough to declare a culprit guilty, instead humans need to be able to understand why this decision was taken. Recently, a number of approaches have been proposed, to provide reasoning for neural network decisions [52,53,54,68]. These solutions introduce additional computations that may need to be executed on the same devices, at the same time, as the main decision taking. Many approaches in this area even rely more or less on the backpropagation algorithm that is also used for training models. CalcGraph allows the user to define not only differential programming layers but also additional pre- and post-processing steps, that need to be performed. Every operation defined for a CalcGraph is considered as parts of the model that will be compiled, which allows to tailor the whole solution process to the underlying hardware resources.

2.4 Model exploration

Neural networks are powerful, generic, and versatile tools to learn relationships between inputs and outputs. However, this versatility and generic problem-solving capability comes at a high price: there are lots of possible configurations to build a neural network. While in theory it should be possible to use machine learning models like black boxes, *i.e.,* without having to know the internals, in practice machine learning is still hard to use with its thousands knobs and parameters and most of the time default settings do not work. Instead, data scientists need to evaluate different models and tune the numerous parameters based on their assumptions and experience against concrete problems and training data sets. In the following, we present the main categories of model exploration and the approaches that have been proposed in literature to automate this process as much as possible.

Feature engineering is the process of extracting features from raw datasets. It is an important step in machine learning. Bishop defines a feature as *"an individual measurable property or characteristic of a phenomenon being observed"* [12]. It can make the difference between a good model and a bad model. Manual feature engineering is a difficult and time consuming task. It requires domain knowledge, data science expertise, and usually a lengthy process of trial and error. *"Automated feature engineering"* or *"feature learning"*, therefore aims at automatically discovering the needed features to build better predictive models faster. Straightforward approaches consists of simply "trying" different features, either exhaustively, randomly, or driven by some form of greedy algorithms, and then compare the results.

Hyperparameters are the parameters used to control the learning process itself. Accordingly, *"hyperparameter optimization"*, sometimes also called *"tuning"*, is the problem of choosing a set of (preferably optimal) hyperparameters for a specific learning algorithm. Besides a manual search, the traditional way of hyperparameter optimization is simply an exhaustive search of a manually specified subset of the hyperparameter space of a learning algorithm. This is called *"grid search"*. To address the obvious performance drawbacks of this approach, different alternatives exist. *"Random Search"* replaces the exhaustive enumeration of all combinations by selecting them randomly. In case of neural networks, this is often more efficient than grid search [11]. Random search can be further improved with adaptive resource allocation and early-stopping, like predefined iterations, data samples, or features and allocate this to randomly sampled configurations [41]. Another option is to use a greedy approach. Other approaches are based on *"Bayesian optimization"* in order to build a probabilistic model of the function mapping from hyperparameter values to the objective evaluated on a validation set. Bayesian optimization aims to balance exploration and exploitation [35, 55].

"Neural architecture search (NAS)" is a recent development that aims to remedy the difficulties of manually designing neural network architectures [70] by automating this process. Most NAS implementations work as depicted in Figure 3. They start by a definition of possible *"building blocks"* that can be used to create the candidate neural network. For example, the state-of-the-art NASNet paper [70] suggests for an image recognition neural network blocks like *identity*, *1x3 then 3x1 convolution*, *1x7 then 7x1 convolution*, etc. A controller (often a neural network itself, e.g. a recurrent neural network) then generates based on these building blocks a candidate neural network. This network architecture is then trained and evaluated on a held-out dataset, which is usually

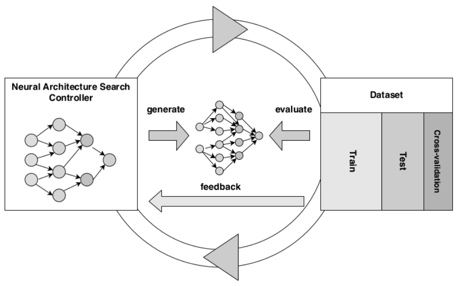

Fig. 3: Neural Architecture Search

divided into a train, a test, and a cross-validation part. The results are used to update the controller and new candidate neural networks are generated. The goal is to generate over time better and better networks. Depending on the requirements and optimization goals (predictive performance, memory consumption, model size, inference time, etc.), various different approaches have been suggested in recent years with a focus on evolutionary approaches [20, 42] and reinforcement-based approaches [69].

The goal of *ensemble learning* is to learn a set of individual models and later use them in tandem [24]. The underlying principle used is that, on average, the joint decisions of a group of models are more accurate than that of the individual models. The key to employ succesfull ensembles is to train a set of diverse and complementary models [64]. The individual models of an ensemble can for example be trained by alternating hyperparameters [64] or using cyclical learning-rate schedules [32].

The model exploration schemes that are detailed above can be performed manually or implemented in a (partially) automated fashion. No matter how the exploration is performed, our CalcGraph solution provides a crucial utility to support the search process. In a process that we coin as model switching, we can move between various pre-compiled models efficiently while assuring that the constraints on computational resource consumption are adhered throughput the exploration.

3 The CalcGraph model

In this section, we detail our proposed CalcGraph, which has been specifically designed to provide control over the computational resource consumption of differential programming. Figure 4 gives an overview of CalcGraph's technology stack. The main elements discussed in this article are depicted in light grey. The CalcGraph consists of a meta-model, model compiler, model interpreter, and heap (referred to hereinafter as CalcHeap). The model compiler takes a model as input, precomputes the heap partitioning, derives an optimal batch size, and generates byte code for the model interpreter. The model interpreter executes the generated byte code. As mentioned in Section 2.1.2, we

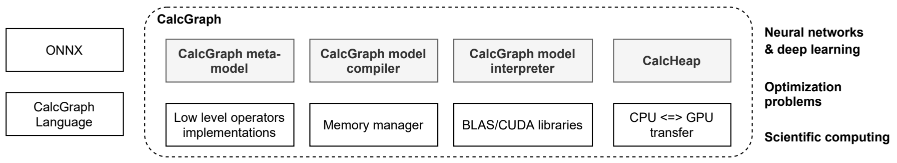

Fig. 4: Overview of the CalcGraph stack

provide our own modeling language, however, since this language is not important in the context of this article, we consider it as out of scope. Additionally, we aim at supporting ONNX[1] models. ONNX is a de-facto standard open format to represent differential programming models. Supporting the operators and building blocks, defined be ONNX, will make it easier for new and experiences developers to get started with CalcGraph. While we focus on neural networks and deep learning, it is important to mention that a CalcGraph, as an abstraction of differential programming layers, can also be used in the context of general optimization problems and scientific computing (see also Section 2.1).

Details of the CalcGraph wil be given in the follwing Sections, starting with the meta-model in Section 3.1, followed by the compiler in Section 3.2 and the interpreter in Section 3.3. Afterwards we will summarize the main features of CalcGraph in Section 3.4 and provide an example in Section 3.5.

3.1 The CalcGraph meta-model

Figure 5 shows the CalcGraph meta-model. The main elements are again depicted in light grey. It is important to notice that a CalcGraph is essentially a lightweight descriptor of a complex computation (learning). This is similar to the computational graph of TensorFlow [6].

Each **CalcGraph** defines an execution mode, which specifies where the graph should run on. This indicates the resources and libraries which are later needed for the execution. Possible execution modes are: CPU(s), GPU(s), or BLAS libraries. Each CalcGraph contains one or many paths.

A **Path** defines a computational sequence of operators. Each operator has a number of input and output variables that are used during execution. Paths specify a calculation mode. Possible values for the calculation mode are *ForwardOnly* (no backpropagation, *i.e.,* no learning involved) and *BackwardEnabled* (forward and backpropagation, *i.e.,* learning involved). For instance, we can create a "pre-processing path" to specify what operators need to be executed on an image before we execute another path for the neural network. The pre-processing path would be in *ForwardOnly* mode, since it is a sequence of operators with no weights to learn. Then, we could define a "neural

[1] https://onnx.ai/

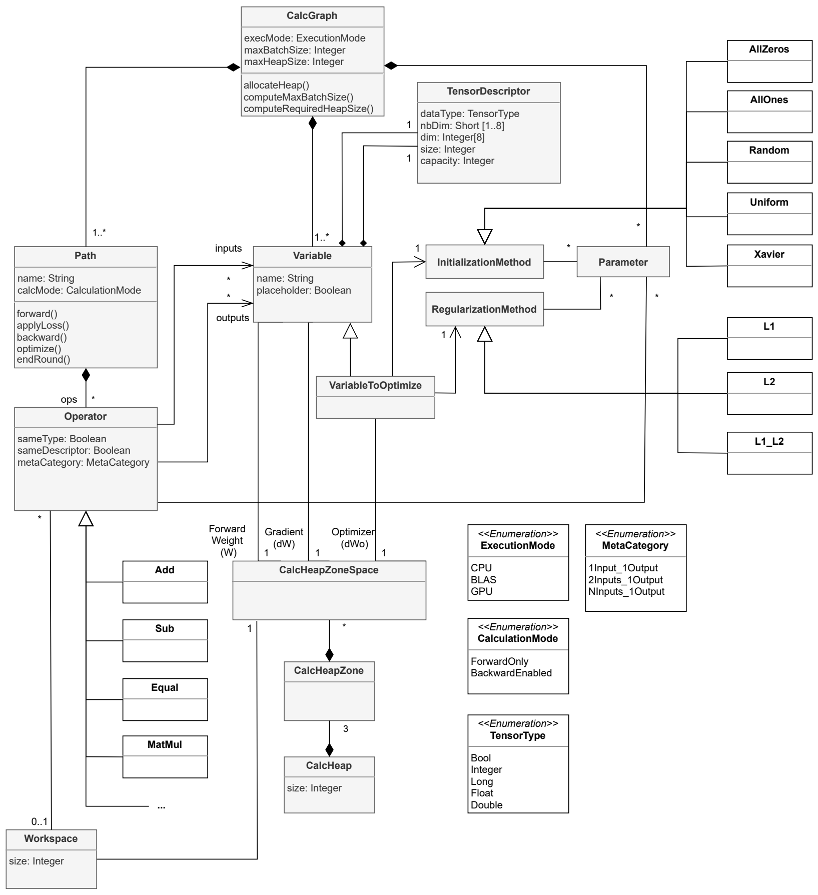

Fig. 5: CalcGraph meta-model

network path" (configured in *BackwardEnabled* mode) with several operators and weights to learn. Last but not least, we could define a "post-processing path" (*ForwardOnly* mode) that converts the output of the neural network to an appropriate format for an application. In the same fashion, we could implement recent XAI methods which investigate the feature maps produced during forward computation [52, 53, 54] or the gradients computed via back-propagation [52, 68], either as stand-alone paths or by adding the necessary operations to the "neural network path" Another example of using multiple paths is a Generative Adversarial Network (GAN): we could configure one path to represent the generative network and another path to represent the discriminative network. In other words, a CalcGraph is a composition or sequence of paths, which define a control flow. In Section 3.5, we present an example of how to define a CalcGraph and declare a control flow using paths. That each path, and with this each operator, has a forward and backward mode (*i.e.,* can be learned) makes the connection to differentiable programming (see Section 2.1). This allows to model differentiable programming layers and therefore deep neural networks in CalcGraph We separate the functions *forward*, *applyLoss*, *backward*, and *optimize* instead of having one function combining all of these. This gives us full flexibility on the batch size and the way we process batches. While bigger batch sizes are more likely to converge, they require lots of rounds to converge. On the other side, very small batch sizes have a risk to not converge but need less rounds until they converge. Finding an ideal (mini) batch size is therefore a trade-off and can be considered as a hyper-parameter. Furthermore, while certain learning problems and datasets might suggest a certain batch size (for some optimizers the learning rate gets harder with increasing rounds), there can be technical limitations why this batch size is not feasible, *e.g.,* available GPU memory or I/O block size. This means, we have to distinguish between the batch size coming from the learning problem and the technical batch size limitations. By splitting the *forward*, *backward*, and *optimize* methods and having an additional method *endRound* (to indicate the end of a training round), we are able to respect the learning batch size despite technical limitations by splitting the batch size in two concepts.

Operators represent atomic operations that can be executed on variables. Examples are *Add*, *Sub*, and *MatMul*. Most operators require one or several inputs and have one output. Others need additional parameters. Operators can be grouped into classes of similar characteristics. One class is the group of *unary* operators that take one tensor as input and produce one tensor of the same size and dimension as output. An example for an unary operator is $Y = sin(X)$. Another class of operators takes two inputs and produces one output, all of the same size and dimension, like $C = add(A, B)$. Then, we have a more advanced class of operators, which needs additional parameters and have more complex dimension relationships between their inputs and outputs. Examples for this class are *MatMul*, *convolution*, and *de-convolution*. We define a set of meta categories (**MetaCategory**) where most operators belong to. These do not need special checkers or implementations regarding size, type, and memory requirements. They are usual linear in terms of memory consumption. Other operators need to provide their own checkers of size, type, and memory needed for the output. The list of all operators can be seen as an instruction set, which defines the possible operations that can be performed on variables with a CalcGraph. This is somehow similar to the x86 instruction set, which defines which operations can be executed on a compatible microprocessor. A de-facto standardized set of operators has been defined by ONNX [3] and we aim at implementing a superset of these operators to be compatible with ONNX models. However, to date we only cover around 1/3 of the most used operators.

Operators have an optional **Workspace**. This is necessary because some operators need additional memory. The required memory is different per target device (CPU/BLAS/GPU). This is a main reason why a change of the target device makes it necessary to recompile the model. For CPU only we do not need a workspace, but for BLAS and GPU it is required. All operators in the whole

CalcGraph model share the same workspace. The size of the workspace is based on the operator with the biggest memory requirement. The workspace is a special zone in the heap.

Variables are the central elements of a CalcGraph. Each variable represents a tensor. There are three types of variables: *1) placeholders* which hold data from outside of the CalcGraph, *2) variables to be optimized*, meaning they have to be updated after a learning operation, and *3) generic variables*, which are usually just the results of operators. For example, the following equation $O = W * I$ shows these three types of variables. I represents a tensor input or a placeholder provided from a dataset outside the CalcGraph. W can be marked as to be optimized, it represents the weights to learn, thus has to be initialized in some way, then updated regularly after each learning round. O is just calculated straightforwardly from the execution of the operator *multiply* on W and I. Placeholders have getters and setters while variables and variables to optimize have only getters (for the value and for the gradient). Variables that are marked as to be optimized require two additional parameters: one to define how to initialize them (**InitializationMethod**, *e.g.,* everything to zero, random, etc.) and one to define what regularization method (**RegularizationMethod**, *e.g.,* L1, L2, etc.) has to be applied on them during the optimization phase. Every variable has two **TensorDescriptor**s. One specifies the generic shape of the tensor and can contain a variable size dimension. The other one represents the current shape of the variable as configured to run with the current batch size. All variables allocate a forward weight in a so-called **CalcHeapZone**. If the path is configured for backpropagation, all variables in that path have in addition to the forward weight also a gradient weight. The variables marked as to be optimized have an extra place for the optimizer.

The **CalcHeap** is an abstraction responsible for allocating memory for the execution of the CalcGraph. It is subdivided into four memory zones: the *forward zone*, the *gradient zone*, the *optimizer zone*, and the *workspace zone*. Each variable in the CalcGraph points to its space, the **CalcHeapZoneSpace**, within one or several of the zones. Variables participating in any path marked for backpropagation will have memory allocated in both forward and gradient zones. Variables marked to be optimized will have in addition space in the optimizer zone. The remaining variables participating in forward only paths will only have memory allocated in the forward zone. Each of the zones is a continuous memory block. As we will see in the following sections, this division of zones is a key characteristic of our CalcGraph approach.

3.2 The CalcGraph model compiler

The compiler takes a CalcGraph model as input. In the model, either the maximum allowed memory size (main and GPU) must be specified, or the maximum allowed batch size, depending on what the compiler should optimize. In case the (optimal) maximum batch size is known, *e.g.,* due to the dataset

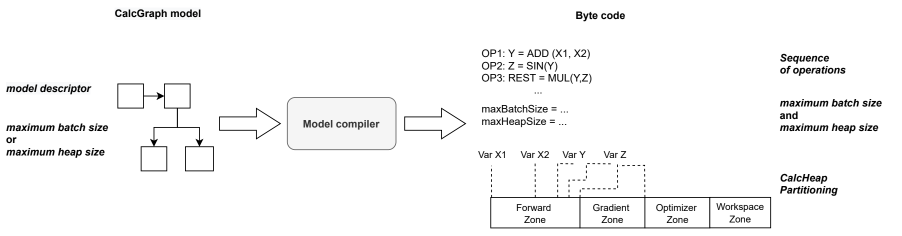

Fig. 6: CalcGraph compiler

or the given learning problem at hand, the compiler can automatically derive the required memory size for this batch size and for the target device. The other way around, the maximum allowed memory size can be specified and the compiler can automatically derive the maximum possible batch size given the memory restrictions. The compiler then generates the byte code of the model and derives the heap memory partitions, to maximize the usage of the allocated heap space, as output. This information is directly stored in the byte code of the model. The guarantee on memory consumption that our compiler provides is a key enabler for applications with strict resource constraints. The compilation process is depicted in Figure 6.

While computing the required memory size from a given maximum batch size is straightforward—with the help of the tensor descriptors, we know the exact size of each variable (see Section 3.1)—computing the maximum batch size based on a given maximum memory size is more difficult. This is because some operators are not linear in their memory allocation. Another reason is that the same operator can have different implementations according to the target device or according to the input tensor size (due to optimizations).

We decided to use a *dichotomic search* algorithm. The base idea is first to start with a batch size of *one* and compute based on it the memory needed. If the memory needed for a batch size of one is already more than the maximum allowed memory, the search stops with an error message. In a future work we plan to investigate how a single CalcGraph descriptor could be split or how we could apply methods like gradient-checkpointing [18] to overcome this limitation. However, CalcGraph models are just small descriptors so that even in case of very limited hardware resources this is only a rather small limitation. If the batch size of one passes the maximum allowed memory test, the next step is to consider if the operators will scale in memory in a linear way. So we compute what the batch size would be if we divide the maximum allowed memory by the memory required for a batch size of one. Next, we increase or decrease the batch size step by step. When we overshoot the maximum allowed memory size we decrease the batch size, if we can still fit more in the maximum allowed memory, we increase the batch size accordingly. We continue this process till we reach an ideal batch size, where even a *+1* on it would overflow the maximum allowed memory. Algorithm 1 shows the pseudo code for this algorithm. We omitted the description of the method *requiredMem*, as it is a straightforward computation of the size for a given batch size (a simple iteration over the operators and variables).

Algorithm 1 Dichotomic search

```
procedure $getMaxBatchSize(maxMem)$
    $startSearch \leftarrow 1$
    $startSearchSize \leftarrow requiredMem(startSearch)$
    if $startSearchSize > maxMem$ then
        $throwException("Insufficientmemory")$
    end if
    ▷ we assume a linear memory requirement first
    $endSearch \leftarrow maxMem/startSearchSize$
    $endSearchSize \leftarrow requiredMem(endSearch)$
    ▷ double the upper bound as long as it is less than $maxMem$
    while $endSearchSize <= maxMem$ do
        $endSearch = endSearch * 2$
        $endSearchSize = requiredMem(endSearch)$
    end while
    ▷ $startSearch \leq maxMem; endSearch > maxMem$
    do
        $x \leftarrow (startSearch + endSearch)/2$
        $xSize \leftarrow requiredMem(x)$
        if $xSize \leq maxMem$ then
            $startSearch \leftarrow x$                ▷ update lower bound
        else
            $endSearch \leftarrow x$                  ▷ update upper bound
        end if
    while $(endSearch - startSearch) > 1$ ▷ reduce search interval while $boundary > 1$
    return $startSearch$             ▷ max batch size is stored in $startSearch$
end procedure
```

The use of CalcGraph models as abstractions allows us to optimize the generated byte code and hence, reduce the computational resource consumption. First, the generated byte code consists of a simple sequence of operations without branches, *i.e.,* the byte code is *branchless.* Although modern processors are equipped with sophisticated branch prediction algorithms, each wrong prediction hits the performance quite substantially. Branching to an unexpected location makes it necessary to flush the pipelines, pre-fetch new instructions, etc. To avoid such dreadful stalls, our generated byte code is branchless. This optimization is possible by analyzing the model abstraction, or more precisely, the paths and operators of a CalcGraph model before generating the byte code. Each operator of the model abstraction is mapped to a specific byte code that can afterwards be directly accessed via *goTo* calls instead of a sequence of branch decisions. This scheme is used and proven to be efficient with other compiled programming languages like Lua [4]. Another optimization that we perform is to merge several operators or to split an operator into several others in case it is beneficial for a target device. For instance, CUDA libraries provide an optimized implementation for linear, recurrent, and convolutional layers. In these cases, there is no need to run first a *Multiply* and then an *AddBias* operator, since we can fuse these two operators into one *LinearLayer* operator. In addition to that, the model compiler can optimize for the data transfer necessary between CPU and GPU, by carefully splitting the model into chunks of independent operations to enable data transfer in parallel with model execution on GPU. This is possible thanks to the asynchronous transfer and execution API of many GPU providers.

Besides the sequence of operations, the model compiler also generates the partitioning of the heap. The heap gets partitioned into four distinct zones: *the forward zone, the gradient zone, the optimizer zone, and the workspace zone.* As mentioned previously, for paths in forward only mode variables only need space in the forward zone, while for paths in backpropagation mode variables need space in the forward zone and in the gradient zone. Variables to optimize need additional space in the optimizer zone. The workspace zone is a common workspace for the whole model and is used in case some operators need to allocate additional memory for some target devices, *i.e.,* BLAS and GPU. It is important to notice that all of these zones are continuous. Each variable of a CalcGraph model points to a distinct position inside of the zones. This is implemented using offsets, *i.e.,* for each variable and for each zone the variable needs space in, the byte code of the CalcGraph model contains an offset specifying the position inside a zone. This offset mechanism allows to omit bound checks during runtime and therefore to further increase performance.

When the model compiler is calculating the offsets of variables in each zone, it always considers the biggest batch size possible within the maximum memory bounds. Later, during the execution time, it is always possible to use a batch size smaller than the maximum allowed one. This will only have the effect of a sparse memory usage, but it is still much cheaper than to free, recompute, recompile, and re-allocate the memory. The following example shows when changing the batch size would be necessary: Let us consider we have a dataset of 1065 entries, and within the memory constraint, we can only fit a maximum batch size of 100. Then, we can proceed by splitting the dataset into 10 mini batches of 100 each and the last one of 65. There is no need to reallocate the memory for the last mini batch. A sparse usage is absolutely fine for such cases.

As mentioned earlier, the model compilation is different for each target device. This allows special memory and operator optimizations per target device. The instruction set and the logic of operators are different on CPU and GPU. While CPUs are optimized for sequential operators, GPUs are best suited to parallelize the implementation of the operators, thus might have different memory requirements. This is why the workspace zone is needed.

3.3 The CalcGraph model interpreter

The model interpreter is a virtual machine. It reads the byte code generated from the model compiler and executes it. As explained in the previous sec-

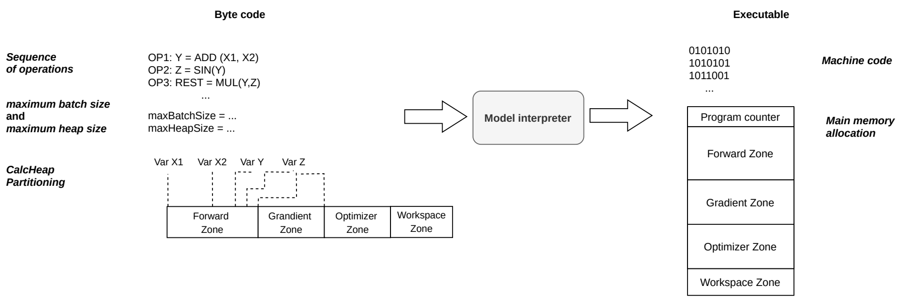

Fig. 7: CalcGraph interpreter

tion, the byte code is a sequence of executions. During execution, the model interpreter takes advantage of the memory segmentation that has been precomputed by the model compiler. This allows the model interpreter to statically allocate the whole required memory for the execution of the CalcGraph model in advance. This avoids expensive memory allocation operations (malloc) during the execution of CalcGraph models and significantly improves the performance. Moreover, it allows to execute CalcGraphs with constant memory usage, no additional memory needs to be allocated nor freed. Therefore, unlike in many other approaches, it can never happen that training a model ends in an out of memory error after hours or days of computation. The working process of the interpreter is depicted in Figure 7.

The execution granularity is per operator. In future work we plan to investigate what advantages and disadvantages another granularity, *e.g.,* per path would have. At the beginning of the execution the pre-allocated memory gets initialized. Initializing just means the memory is set to 0, *i.e.,* it is just a single *memset* operation. During execution the place of each variable inside this memory block is defined by the offset that has been precomputed by the model compiler and stored inside the byte code. The fact that each variable has a precomputed static place inside the memory allows to completely omit bound checks during runtime, which again increases the performance. In addition to the four zones (forward, gradient, optimizer, and workspace), the main memory also holds a *program counter*. Besides the fact that paths can be executed in forward and in backward mode, and that therefore the program counter cannot be just increased but must also be decreased, our program counter is a standard program counter. The way we divide the main memory and generate the byte code from the model makes it unnecessary to use an additional stack during execution.

The common way to train models is to allocate and free memory dynamically during learning, and this per model. This is not only extremely costly but the required main memory for training is not known in advance, possibly leading to out of memory errors after hours or days of computation. It makes it also more difficult to optimally utilize the available resources. One of the core concepts behind our CalcGraph model is that multiple models can share

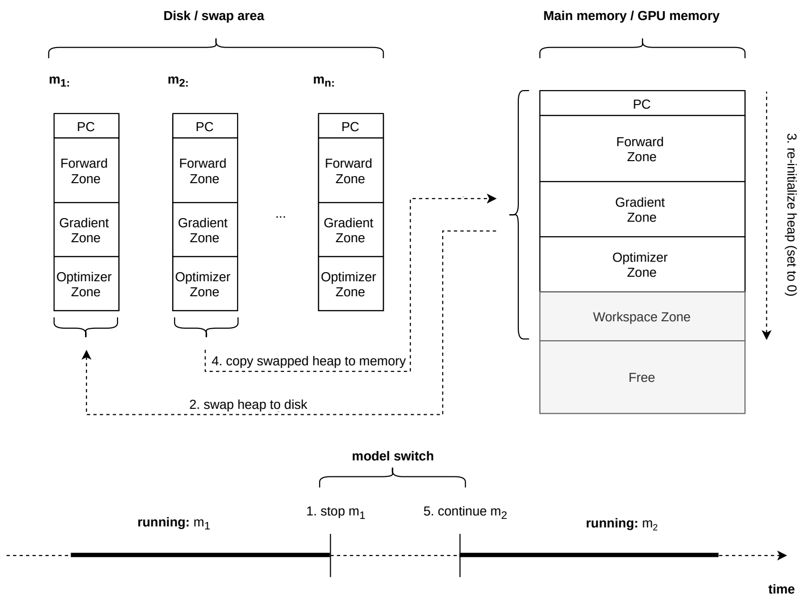

Fig. 8: Process of switching a model

the same pre-allocated memory and just repartition it, instead of expensively freeing and reallocating memory. As a reminder, repartitioning the CalcHeap means just initializing it to 0. In reference to the concept of context switching, we refer to our approach of switching models into the pre-allocated memory as *model switching*. Just like a context switch is a procedure to change from one task (or process) to another while ensuring that the tasks do not conflict, a model switch is a procedure to change from one model to another while ensuring that the models do not conflict. And just as effective context switching is performance critical to provide user-friendly multitasking, effective model switching is performance critical when it comes to executing many CalcGraph models, as it is the case for hyperparameter or neural architecture search. A model switch consists of the following basic operations:

- stop the execution of the current model
- swap the heap of the current model to storage
- re-initialize the heap, to avoid leaks from prior executions
- copy the swapped heap of the new model from storage to memory
- continue the execution of the new model

These steps are depicted in Figure 8. Since each zone is a continuous block, swapping the heap from storage to main memory and vice versa is very cheap because we only need a single memory copy (*memcpy*) operation per zone. Furthermore, it is not always necessary to swap all zones. As soon as the variables to optimize are optimized, there is no need to keep the gradient zone, *i.e.,* the gradient zone will no longer be swapped. Similar, if the target device does not require a workspace, there is no need to swap this zone.

Our approach of inexpensive model switching is very beneficial for many practical learning problems using multiple models. This is especially the case for different instances of the same model, or different parameters for the same model. For example, recommender systems need to train at least one model per user, or per category of users. Exploration with various models an architectures (e.g. different number of layers) would lead to additional effort during the compilation phase. But even in this case, switching between the pre-compiled models is efficient as we only require a single *memset* on the allocated area and one *memcpy* per zone. Without an effective model switching mechanism, one would need to costly allocate and free the memory dynamically during learning–and this per model.

3.4 CalcGraph summary

From the perspective of the CalcGraph as a model (of neural networks) the CalcGraph can be considered as an abstraction over the generated source code that is executed. The CalcGraph model, for example, abstracts from the fact that generated byte code is branchless, from different optimizations for GPU/BLAS/CPU, from any platform specfics etc. From the perspective of the process from a neural network to the CalcGraph, it, or more specifically the **Paths** and **Operators**, can also be regarded as a decomposition. The main features of our CalcGraph approach can be summarized as follows.

– We compile each CalcGraph before execution which gives us full control over the memory consumption.
– We can either choose to set an upper limit and select the batch size accordingly or use the batch size as an input to derive the memory requirements of the full process.
– The generated byte code is branchless which assures that the execution runs without the risk of slowdowns caused by wrong branch prediction.
– The used memory (the CalcHeap) is split into zones (*forward*, *gradient*, *optimizer*, *workspace*) and the position of each variable within these zones is given by an offset which allows for fast access and omits the need for bound checks.
– The CalcHeap as well as the individual zones are allocated statically to avoid expensive memory allocation operations during model execution.
– Lastly, this scheme also allows us to switch between individual precompiled models efficiently as we can reinitialize blocks or even the full CalcHeap in one go and re-partition the memory according to the new compiled model.

The deterministic and yet flexible memory allocation as well as the runtime optimizations, present CalcGraph as a perfect solution for applications with strict resource constraints such as edge-AI. Furthermore, the ability to switch efficiently between several precompiled models, provides a solid foundation for model exploration.

3.5 A CalcGraph example

In this section, we show a practical example on how to use our CalcGraph to train a simple linear model: $O = W * I$. Algorithm 2 shows the declaration of a CalcGraph. First, we create a CalcGraph, then we configure the input and output variables as two placeholders. These store the data from the dataset. In a second step, we create a variable W marked as "*to optimize*", and specify its initialization method as a uniform distribution. Finally, we configure the variables' descriptors by stating the data type and dimensions of the tensors. We use 0 to indicate that this dimension can hold up a variable batch size of dimensions (depending on the available memory at runtime). In short, in this example, we are converting a 6-inputs problem to 3-outputs using a linear relation, regardless of what the batch size will be.

Algorithm 2 CalcGraph declaration

```
1: $calcGraph \leftarrow CalcGraph()$                                        ▷ init new Calcgraph
2: $I \leftarrow calcGraph.newPlaceholder()$                                 ▷ create placeholder vars
3: $O \leftarrow calcGraph.newPlaceholder()$
4: $W \leftarrow calcGraph.newVarOpt()$                                      ▷ create optimizable var
5: $W.initUniform(min = 0.1, max = 0.9)$                                     ▷ init $W$ uniform
6: $calcGraph.configureVar(I, datatype = Double, size = [0, 6])$             ▷ batchSize * inputs
7: $calcGraph.configureVar(O, datatype = Double, size = [0, 3])$             ▷ batchSize * outputs
8: $calcGraph.configureVar(W, datatype = Double, size = [6, 3])$             ▷ inputs * outputs
```

Next, in Algorithm 3 we define the operations to be done on the CalcGraph. To show an example of multi-path usage, we create two paths. The first one is a learnable path that contains the main sequence of operations: multiplication between W and I, subtraction with the output, and an absolute value as a loss metric. The second path is an additional post-processing path that calculates RMSE as an extra metric. Again, in this example, we show how a very simple linear model can be defined. For now, we have not implemented the logic needed for more advanced convolutional models. However, after the implementation is done, creating a more recent deep neural network model like e.g. ResNet [29] would be possible in the same way as portrayed in Algorithm 3, with the only difference being additional appended steps for a path and new available functions to declare layers like convolution or max-pooling.

After the declaration of variables, paths, and operations is done, we allocate memory through the CalcHeap, associate it to the current CalcGraph and compile the model. As described in Section 3.2, we can either choose to allocate a fixed memory size and derive the maxmimum batch size for this (Algorithm 4) or we give a target batch size as input and compute the required memory size (Algorithm 5).

The last step is to get the dataset, move it to the CalcHeap, run a forward and backward on path 0, and a forward on path 1. Finally, we can get the RMSE metric result. This is shown in Algorithm 6.

Algorithm 3 Paths declaration

```
Require: $calcGraph$ initialized
 1: $p_0 \leftarrow newPath(backward = True)$    ▷ create new path with backward enabled
 2: $p_0.addStep(Y, matmul(I, W))$    ▷ $Y := I * W$
 3: $p_0.addStep(D, sub(Y, O))$    ▷ $D := Y - O$
 4: $p_0.addStep(E, abs(D))$    ▷ get absolute error
 5: $p_0.setOptimizer(sgd(learningRate = 0.001))$    ▷ set optimizer for path $p_0$
 6: $calcGraph.addPath(p_0)$
 7: $p_1 \leftarrow newPath(backward = False)$    ▷ create new RMSE path with backward disabled
 8: $p_0.addStep(R, rmse(Y, O))$    ▷ $R := rmse(Y, O)$
 9: $calcGraph.addPath(p_1)$
```

Algorithm 4 CalcHeap declaration for given max memory size

```
Require: $calcgraph$ and paths initialized; $memSize$; $targetDevice$
 1: $calcHeap \leftarrow CalcHeap()$
 2: ▷ compute max $batchSize$ for $memSize$
 3: $batchSize \leftarrow calcGraph.computeMaxBatch(memSize, targetDevice)$
 4: $calcGraph.compile(batchSize)$    ▷ compile $calcGraph$ with $batchSize$
 5: $calcHeap.allocate(memSize)$    ▷ allocate memory
```

Algorithm 5 CalcHeap declaration for given batch size

```
Require: Calcgraph and paths initialized; $batchSize$, $targetDevice$
 1: $calcHeap \leftarrow CalcHeap()$
 2: ▷ compute memory requirement for $batchSize$
 3: $neededHeap = calcGraph.computeMemory(batchSize, targetDevice)$
 4: $calcGraph.compile(batchSize)$    ▷ compile $calcGraph$ with $batchSize$
 5: $calcHeap.allocate(neededHeap)$    ▷ allocate memory
```

Algorithm 6 Running the CalcGraph

```
Require: $calcGraph$ compiled; $calcHeap$ allocated; $input$ and $target$ tensor
 1: $timestamp \leftarrow getCurrenTime()$
 2: $calcGraph.setSeed(timestamp)$    ▷ set random seed for learning
 3: $calcGraph.initialize(calcHeap)$    ▷ set all zones and variables
 4: ▷ One round of learning on $p_0$
 5: $calcHeap.set(calcGraph, I, input)$    ▷ populate $I$ with $input$
 6: $calcHeap.set(calcGraph, O, target)$    ▷ populate $O$ with $target$
 7: $calcGraph.foward(p_0)$    ▷ run forward on $p_0$
 8: $calcGraph.backward(p_0)$    ▷ run backward on $p_0$
 9: ▷ forward on $p_1$ to get rmse
10: $calcGraph.forward(p_1)$    ▷ run forward on $p_1$
11: $rmse \leftarrow calcGraph.getVar(R)$
```

4 Evaluation

This section reports on the experiments we performed to assess the capabilities of CalcGraph comparing it against two state-of-the-art solutions: TensorFlow (TF) and PyTorch.

4.1 Experimental setup

For all experiments, we either report on the main memory consumption, GPU memory consumption, or execution time as key performance indicators.

We use the MNIST dataset[2] for all our experiments. MNIST is a large database of handwritten digits that is often used for training various image processing systems and for training and testing in the field of machine learning. More specifically, we use the greyscale colors from 0-255 in 28x28 pixels for our experiments. This results in 784 inputs (28x28 pixels) and 10 outputs (one for each digit between 0 and 9). The goal is to train a machine learning model that can accurately detect the correct handwritten digit. The MNIST dataset is a well-known and often used dataset for machine learning experiments. It is limited in scope, results are known and therefore allows us to focus on performance evaluation and use the accuracy as a crosscheck criteria.

For all the approaches, we train a neural network with the same architecture: three dense layers, fully connected, using a bias. The first layer converts the 784 inputs to 64 outputs, uses a bias and a sigmoid activation function. The second layer maps the remaining 64 inputs to 64 outputs, uses again a bias and a sigmoid activation function. Finally, the third layer converts the 64 inputs to the expected 10 outputs, uses a bias and a softmax to get the probabilities and categorical cross entropy loss. Adam [36] is used as optimizer. We use a dataset with train = test = 10,000 images. This means we have a total of 10,000x784 inputs. For all approaches we force a transfer to GPU after each round. This can be optimized depending on the batch size, but for simplicity reasons we emulate here a transfer after each round.

We executed each experiment 20 times to assess the reproducibility of our results. Unless stated otherwise, all the reported results are the average of the 20 executions and all experiments have been executed on a computer with an Intel Core i7-7820X CPU at 3.60Ghz with 16 cores, 64GB RAM, and a SSD as storage. We used a Nvidia GeForce GTX 1060 with 6GB RAM. As OS, we use Manjaro Linux (an Arch Linux derivate) in version 20.0.3. We use TensorFlow (GPU) in version 2.2.0, without eager execution and fully compiled. For PyTorch we use version 1.4.0. We use Python in version 3.8, CUDA in version 10.1.243 and the cudnn library in version 7.6.3. After each experiment we restart the computer to avoid any side effects.

We share the source code[3] to reproduce our experiments on TensorFlow and PyTorch. Although we advocate the open science principles, we do not intend to share the source code or executable of CalcGraph at this time. We are currently working on a licensing model of CalcGraph, which is a commercial project at its core. Nonetheless, sharing the under- lying principles and demonstrating its optimization potential is of interest to the scientific community.

[2] http://yann.lecun.com/exdb/mnist/
[3] https://hub.datathings.com/papers/2021-sosym-journal

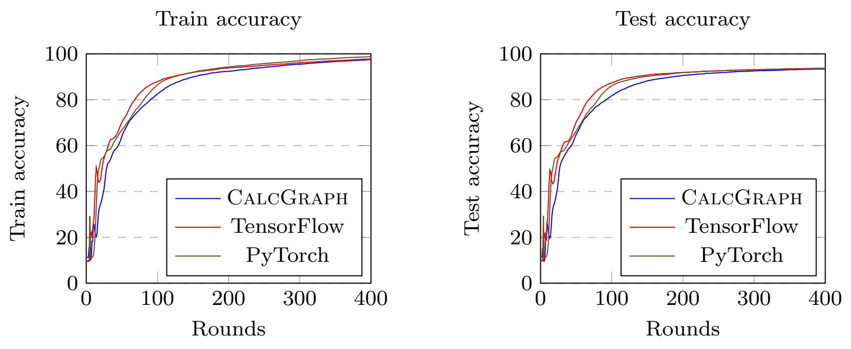

Fig. 9: Single model: Train accuracy (left) and test accuracy (right) of CalcGraph, TensorFlow, and PyTorch.

4.2 Accuracy of a single model

We first evaluate the accuracy of a single CalcGraph model. The accuracy is an important indicator and different approaches are only comparable if they reach a comparable accuracy. In the following, we compare CalcGraph, TensorFlow, and PyTorch. In order to do so, we *train* our model (as described in 4.1) for *400 rounds* and distinguish between the train- and test accuracy. To compare the three frameworks, we fix the batch size to 10,000. For CalcGraph, we could have also decided to fix the size of the CalcHeap and derive the batch size for this size. However, since this is not feasible for the other frameworks, we fix the batch size instead.

Figure 9 shows the train accuracy (left side) and test accuracy (right side) of all three frameworks over the 400 rounds. As the figures show, our implementation of CalcGraph **reaches the same range of accuracy** than TensorFlow and PyTorch. The reason for the slight differences are the random initialization values. If we use the exact same initialization values, we would get the exact same curves. From this experiment, we can conclude that our CalcGraph model is in terms of train accuracy and test accuracy comparable with two state-of-the-art frameworks.

Next, we will evaluate the efficiency of a single CalcGraph model in terms of memory consumption (main- and GPU memory) and execution time.

4.3 Memory efficiency and execution time of a single model

In this section, we evaluate the memory requirements—both main memory and GPU memory—and the execution time for a single CalcGraph model. We use the same experimental setup than for the previous experiment (see 4.1 for the description). We compare the results again to TensorFlow and PyTorch. It is important to notice that we do not measure the time to parse the csv file. In fact, we parse the csv file outside the experiment and store already the tensors in a binary format. We do this, because we do not want to bias the experiment by in fact evaluating the csv parser implementation of the different frameworks. The goal of this experiment is to evaluate how efficient a single CalcGraph model is compared to the state-of-the-art. In this experiment, we train only a single model and therefore not benefit from our model switching mechanism. Instead, we evaluate here the impact of our optimizations regarding memory efficiency and execution time. Namely, we precompile the model before execution, generate brancheless bytecode and set up the memory (CalcHeap) in such a way to reduce lookups and allocations to a minimum. Please refer to Section 3 for details on our optimizations.

We *train* the model for *400 rounds* and we *fix the batch size to 10,000*. Again, for the CalcGraph, we could have also decided to fix the size of the CalcHeap and derive the batch size, but since this is not possible for TensorFlow and PyTorch, we fix again the batch size. A first advantage of our CalcGraph is that the exact needed CalcHeap size is automatically derived *before the execution*. In this experiment, for a batch size of 10,000, we need a CalcHeap of 83MB. TensorFlow allocates by default the whole available GPU memory, in our case around 5GB (available and not used by the operating system). There are several ways to limit this behavior and restrict the amount that TensorFlow can allocate[4]. However, it is difficult to derive the minimum amount that a certain execution requires and can be only determined experimentally for every model and batch size. This means different setups need to be executed until we find the minimum possible configuration which does not fail during execution. This can become quickly a very time consuming task. While it is already difficult for a single model, it is especially problematic in case of model exploration, where we train multiple models with different configurations. Nonetheless, we experimentally evaluated the minimum GPU memory that TensorFlow needs to execute the model. We started with a memory size of 0 and then increased in steps of 10MB until the execution successfully finished. For all experiments, we limit the GPU usage to this experimental minimum. PyTorch always allocates the GPU memory dynamically on demand *i.e.,* we cannot influence the GPU memory allocation.

Figure 10 shows the measured values for CalcGraph, TensorFlow, and PyTorch. There are several important points to note. First, it can be seen that CalcGraph requires a lot less memory—both main memory as well as GPU memory—than TensorFlow and PyTorch. Looking at the execution time, it can be seen that CalcGraph is also faster than PyTorch and TensorFlow. Note that for CalcGraph and TensorFlow, the model is precompiled and the time to compile is **not** included in our experiment. PyTorch models on the other side are dynamically compiled during execution. We can conclude that **for a single model CalcGraph is slightly faster than TensorFlow and PyTorch while using less main- and GPU memory**.

It is important to notice that the total memory needed for our CalcGraph is 247MB main memory and 193MB of GPU memory, although we explained

[4] https://www.tensorflow.org/guide/gpu

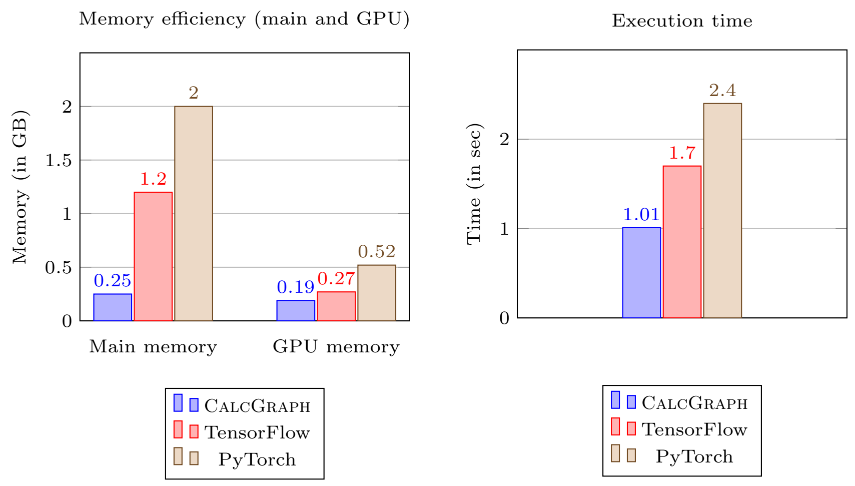

Fig. 10: Single Model: Memory consumption (left) and execution time (right) of CalcGraph, TensorFlow, and PyTorch. Lower values are better.

earlier that the CalcHeap size is just 83MB. As we execute the CalcGraph on GPU, one could expect that the GPU usage should be only 83MB. Indeed, the CalcGraph itself only requires a heap of 83MB. However, the difference of GPU memory, around 110MB, are due to required CUDA libraries and handlers. The same counts for TensorFlow. However, TensorFlow loads around 190MB of libraries and handlers. The actual computational graph of TensorFlow has almost exactly the same size than the one of CalcGraph, 80MB. However, it needs to be evaluated experimentally.

In the next section, we will evaluate a multi-model scenario.

4.4 Multi-model performance

So far, we have shown that even for a single model our proposed CalcGraph is faster than TensorFlow and PyTorch while using less memory resources. In this section, we now switch to model exploration and training multiple models. The goal of this section is to evaluate the effect of model switching (reusing the CalcHeap) on the performance. We simulate a hyperparameter optimization use case and train multiple generated models. We use the same model as in the previous sections (see 4.1 for the description). We fix the structure of the neural network for this experiment and randomly modify learning parameters, optimizers, and initialization methods for the weights. These parameters are not very interesting in the context of this experiment. To be able to compare the different approaches, they just need to be the same along all compared frameworks. The idea is to simulate a hyperparameter optimization scenario as a use-case of model exploration as introduced in Section 2.4. Therefore, the interesting parameters are the number of models to train, and the learning rounds per model. We report on the execution time for different numbers of learning rounds per model (1, 10, and 30) and different numbers of models to train (1, 10, 100, 1000). Like in the previous sections, we compare these values to TensorFlow and PyTorch.

Figure 11 presents the result in form of a scatter chart. The x-axis of the chart shows the number of models to train, multiplied by the learning rounds. This number reflects the computational complexity of a task. The y-axis shows the execution time in milliseconds to finish this task. Both axis use a logarithmic scale. This chart shows the gain of our CalcGraph compared to TensorFlow and PyTorch depending on the number of models and learning rounds. It shows that the gains are bigger for lower total computational complexity. This is not surprising, as it means that the more expensive a computation gets the less impact our fast model switching and optimizations have on the total execution time. Unfortunately, it is not possible to reliably measure the time TensorFlow and PyTorch need for memory allocations when moving between models. However, the literature supports our claim that these allocations can be dreadful and in consequence, developing more efficient allocators is an ongoing task [2,14,22,39]. Our own results show that our optimizations have a significant impact that only becomes negligible when the total computational complexity of a task becomes the dominant factor.

The conducted experiment demonstrates that CalcGraph outperforms TensorFlow and Pytorch for specific use cases of model exploration. One such use case is meta-learning where, except for online learning, 10 to 30 learning rounds are considered as the most relevant scenario [67]. After this amount of rounds we should know if we have already a better model and can stop the exploration of this model. The area highlighted in the figure in blue is what we therefore consider as a “typical” meta-learning case. It is important to notice that this corresponds to one single meta-learning step. Note that these gains are cumulative. The more meta-learning steps we do, the more the benefits even for the worst case scenario running for a long time.

Furthermore, we argue that CalcGraph is perfectly suited for adapting to specific real world scenarios where it can also be assumed that possible models can be quickly filtered within lower number of learning rounds. In addition, in industrial applications, solutions are often replicated for many similar use cases once they work successfully. Here again, fine-tuning models to specific processes should reflect the highlighted area of Figure 11.

4.5 Parallel execution

Knowing the memory requirements of a CalcGraph model in advance allows us to optimize the usage of the available resources by executing as many models in parallel as possible. In this experiment, we report on the number of models that we can execute in parallel. We use the same model than in Section 4.3, but execute it in n parallel processes. The limiting factor is unsurprisingly the available GPU memory, which is 5GB. As we discussed previ-

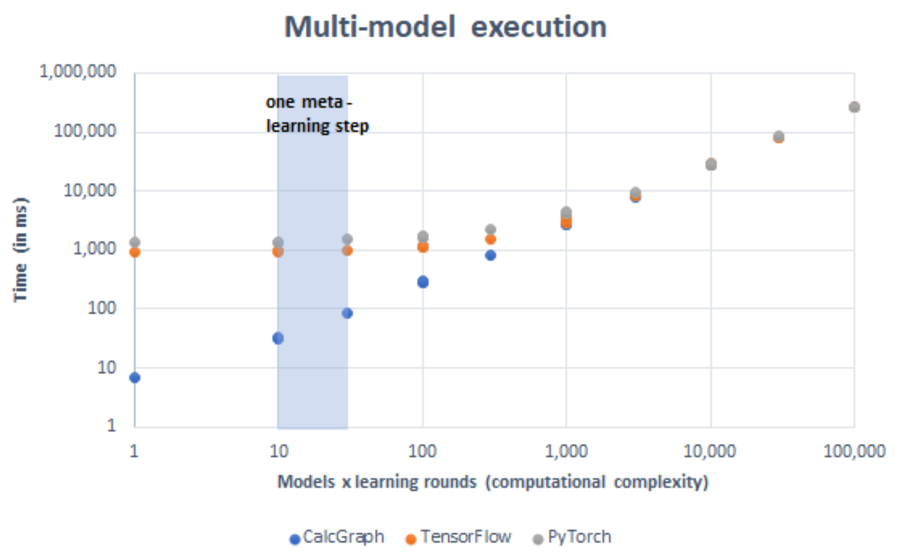

Fig. 11: Multiple models: execution time for a varying computational complexity (number of models multiplied by the the number of learning rounds). Lower values are better.

ously, for CalcGraph, we know exactly how much GPU memory one model needs. Therefore, we can just split the available GPU memory and know in advance how many models we can run in parallel. For TensorFlow the number of models that we can run in parallel depends on how much we limit the GPU memory that TensorFlow is allowed to allocate. In Section 4.3, we experimentally evaluated the minimum GPU memory. As TensorFlow otherwise allocates the whole available GPU memory—and therefore only one model can be executed at a time—we reuse the evaluated minimum size in this experiment to maximize the number of models that can be theoretically executed in parallel with TensorFlow. As a reminder, CalcGraph requires 83MB for the CalcHeap and 110MB for CUDA handlers and libraries. TensorFlow needs 80MB for the computational graph and 190MB for the CUDA handlers and libraries, *i.e.,* around 80MB more than CalcGraph. TensorFlow allocates the handles dynamically and not statically. Only the computational graph is allocated statically. This means, similar to PyTorch, it crashes not in advance but during the execution.

For PyTorch, as GPU memory is allocated dynamically, we can only evaluate experimentally how many models we can execute in parallel. Therefore, we increase the number of models in steps of one, till PyTorch fails to execute them. Figure 12 shows the number of models that we can execute in parallel for the different frameworks.

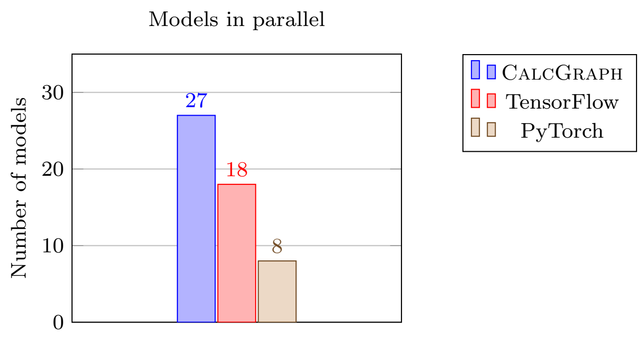

Fig. 12: Number of models that can be executed in parallel. Higher numbers are better.

As can be seen in the figure, CalcGraph allows to execute the most models in parallel on the available resources. This has the potential for huge speedups. The exact factor depends on different parameters. For example, the number of available GPU cores. As most machine learning tasks are build for parallelization, it can easily happen that already a single model saturates all cores on a single GPU. In such cases, executing several models in parallel would have only limited impact. The ratio between CPU- and GPU load can also heavily impact the speedup of parallelization. In many cases, the higher the CPU load, the higher the speedup. This is due to the fact that the GPU processing is already very optimized for parallelization and therefore the GPU cores are more saturated. Whereas CPU cores in modern hardware are often less saturated and can therefore benefit more from the parallelization. Therefore, the higher the CPU load in relation to the GPU load is, the higher we can expect the speedup due to model parallelization. A similar parameter is the ratio between data transfer (between CPU and GPU) and GPU computation. Depending if one takes much more time than the other, the impact of model parallelization is heavily influenced. In future work, we plan to focus on optimizing the parallel computation unit occupancy, *i.e.,* splitting the number of available GPU cores depending on the complexity of the CalcGraph model and statically allocate these cores. To conclude, the potential speedup when executing models in parallel is huge. However, the exact speedup factor depends on many parameters and is difficult to quantify in a general manner.

4.6 Threat to validity

We have decided to evaluate our approach on the well-known MNIST dataset that is often used to evaluate machine learning approaches, *e.g.,* for new neural network architectures. Despite the fact that we showed that our proposed CalcGraph can outperform state-of-the-art frameworks like TensorFlow and PyTorch both in terms of execution time as well as in memory consumption, one threat to validity remains that the evaluation case study might be especially appropriate for the presented solution. Additional case studies might be needed to be considered to better estimate the general applicability of the presented approach. Nonetheless, as we argued, the evaluated case study is representative for the use cases targeted by our approach.

5 Limitations and future work

Our CalcGraph approach and implementation comes with a number of limitations, which we plan to address in future work.

First, as stated throughout this article, our CalcGraph is an abstraction over differentiable programming layers and therefore suited for deep learning using neural networks and other gradient-based optimization algorithms. However, it is not well adapted for other learning algorithms. Nonetheless, it could also be used in the context of general optimization problems, scientific computing, and simulations–where a static memory allocation can also ease the deployment. One track of post-processing, that is of particular interest to us, is to provide explanation for deep neural network decisions. Some state-of-the-art methods in this domain reuse the backpropagation algorithm, that is also used for training models, in some way (e.g. [53, 54]). As we have already implemented backpropagation for CalcGraph, a good first step could be to implement such methods on top of our approach.

As mentioned in Section 3.1, the set of CalcGraph operators defines the instruction set, *i.e.,* the possible operations that can be performed on variables. While our goal is to eventually implement all operators defined in ONNX, some of the operators defined in TensorFlow [5] related to image processing, as well as some additional ones, we have not yet reached this goal. All operators need to be implemented for all data types, for each execution mode (CPU, BLAS, and CUDA), and for each calculation mode (forward and backward). Currently, we only support around 1/3 of the operators defined in ONNX and are therefore not yet fully compatible with ONNX models. This is also the main reason why we limited our experiments to using a simple linear model and MNIST[5] as a dataset. We are not yet able to model more recent convolutional neural networks such as ResNet [29]. As soon as the latter is possible, we plan to perform further experiments on standard benchmarks like ImageNet [50] for vision or BERT [21] for language analysis.

In Section 4 we introduced the hardware setup we used for our experiments. We opted for commodity CPU and GPU that are commonly used in personal desktop setups. In this paper, we focus on domains with constrained hardware capabilities. Hardware specifically designed for edge-AI, like the Nvidia Jetson family [58], would be an extreme example of this and therefore, interesting to investigate in the future. The Jetsons are based on ARM-chipsets which makes it more difficult to port our solution. Therefore, we consider evaluations on that sort of hardware out of scope for this paper but identify running

[5] http://yann.lecun.com/exdb/mnist/

CalcGraph on a Jetson as an important next step. At the moment it is also not possible to distribute a CalcGraph execution over multiple GPUs. However, for the future we would like to allow distributed execution to make applications based on CalcGraph even more flexible and efficient. We expect that the optimization regarding efficiency, that we introduced for single hardware components in this paper, will translate to multiple hardware instances as well.

Another limitation of our current implementation is that the minimum amount of memory required is defined by the size of a CalcGraph with batch size equals to one. In future work we plan to investigate how to overcome this limitation, *e.g.,* by splitting a CalcGraph into several parts. A full CalcGraph could be broken down into one CalcGraph per operator, etc. Another promising approach we want to look into is to remove intermediate variables and recalculate them when needed again. An interesting approach to reduce the memory needed for training deep neural networks is so-called *gradient-checkpointing* [18]. The idea is that the memory expensive part of training deep neural networks is computing the gradient of the loss by backpropagation. This requires to keep the results of the forward pass in memory. However, by checkpointing nodes in the computation graph defined in the model, and then recomputing the parts of the graph in-between those nodes during backpropagation, it is possible to reduce the required memory since not all results in the forward pass need to be kept. However, given the fact that a CalcGraph is only a lightweight descriptor, even on very limited hardware, the fact that this descriptor needs to fit into memory is not a big limitation.

Currently, the execution granularity of a CalcGraph is per operator. While at a first glance this seems intuitive, it means that the ordinal pointer needs to be kept in the CPU for each operator. In future work, we plan to investigate what advantages and disadvantages another granularity, *e.g.,* per path would have. The idea is that with a path granularity, you would only need the ordinal pointer once for the whole path instead of for every operator.

6 Related work

While in our work, we also focus on neural networks, our contribution centers around the idea of CalcGraph as an efficient foundation to build model exploration on top and to contain resource consumption for constrained domains. In this sense, our work is comparable with the computational graph of TensorFlow [6] or PyTorch [45]. But unlike TensorFlow's computational graph, CalcGraph has been specifically designed with model exploration in mind. We leverage model abstractions to generate highly optimized code, predict the required resources in advance in order to optimally leverage the available resources, and enable a way to efficiently switch models. These are key characteristics for efficient model exploration use cases such as meta-learning [8, 60, 61], neural architecture search [70] or ensemble learning [64, 66]. In other words, we focus on the enablers of efficient model exploration rather than on the exploration, *i.e.,* search algorithm, itself. Both, TensorFlow's computational graph and our CalcGraph are computational graph models and both are statically compiled. However, the key difference is the way we allocate and manage memory. First, TensorFlow allocates all available GPU memory (can be limited to a value) at once. While this allows for intelligent caching strategies and some other optimizations, like reduced transfers from CPU to GPU, it makes it difficult to optimally exploit the available resources and thus to execute as many models in parallel as possible. Secondly, TensorFlow allocates the main memory dynamically. This makes it not just hard to know in advance how much main memory will be needed but also makes model switching more expensive. In contrary, CalcGraph is optimized to handle multi-models more efficiently and to optimize the usage of the available resources. By computing main memory and GPU memory beforehand and by statically only allocating what is needed, CalcGraph can optimize the usage of the available resources and increase the number of models that can be executed in parallel. In addition, our proposed model switching mechanism further increases the usage of available resources. PyTorch on the other hand is not statically compiled and allocates memory (main memory and GPU memory) dynamically. While PyTorch is heavily optimized for many deep learning tasks, the fact that memory is dynamically allocated can lead to unforeseen out of memory exceptions during the execution. In addition, not knowing the resource requirements beforehand makes it also more difficult to optimize the usage of the available resources. Thus making it more difficult to optimize it for meta-learning, *i.e.,* multi-model scenarios.

The problem that required computational resources (time and memory) are often not known in advance has been discussed in the literature in different areas. Priya *et al.,* [48] focus their work on machine-learning problems. They propose to apply a meta-learning framework to relate characteristics of datasets and current machine state to the actual execution time. They predicted the time of six classifiers over 78 datasets using four regression algorithms and showed that their approach can outperform commonly used baseline methods. Lee and Schopf [15] also argue that knowing the execution time and the resources usage of long running tasks in advance is very beneficial. Besides avoiding unnecessary waiting times, it may enhance the scheduling and thus minimize the overall execution time. While they focus their work not on meta-learning (or machine learning) tasks but on parallel applications running in shared environments, the principle problems remain the same. They propose to use regression methods and filtering techniques to derive the application execution time without using standard performance models. First results show that their approach delivers tolerable prediction accuracy and can improve the accuracy dramatically by using appropriate filters. Rather than predicting the memory consumption, as the above mentioned works suggest, our proposed approach leverages models as descriptors to precisely compute the memory requirements in advance and then optimally utilizes the available resources. Instead of forecasting that the result will take so and so long and require that much memory, our approach let you specify the time and memory specifications and tries to find the best result respecting these specifications.

The high computational demands of deep learning is a current hot topic of computer science. As the technology is dominating machine learning benchmarks in many domains like image [50] or language [21] processing, it is only logical that we see more and more real world applications being developed. However, the integration is often hampered by high computational demands, with unsuitable memory resources on the target machines or unsatisfactury execution times [44]. Several research directions exist to reduce the resource consumption of deep learning applications, most notably pruning, quantization and knowledge distillation. Network pruning [13,38] tries to identify unimportant parts of the neural network that can be removed with little effect on accuracy, leading to faster, smaller models. Quantization, mergers network weights and values into buckets to move from floating-point to a lower precision representation [23,34]. Moving to integer precision is reported to reduce the memory footprint and inference time up to a factor of 16x. The key idea of knowledge distillation is to use the state of an already trained big network to help to improve the performance of a smaller and faster model [19,30]. Our approach of efficient memory management to improve the resource consumption of differential programming is orthogonal to the just mentioned techniques. This means that in the future, we could potentially use compressions methods to further increase the efficiency of CalcGraph. In addition, the techniques again require some sort of model exploration to determine which parts to compress. This is a task that can be processed more efficiently using our approach.

7 Conclusion

In this article, we discussed taming the high costs of differential programming, with a focus on domains with constrained hardware resources and efficient model exploration. We proposed CalcGraph, a novel model abstraction of differentiable programming layers, which is particularity suitable for deep learning and neural networks. We showed that CalcGraph provides an efficient foundation to built model exploration implementations on top. We leveraged models (CalcGraphs), similar to how some compilers use abstract syntax trees (AST), to drive and optimize the compilation process. More specifically, our article discussed the following three contributions:

– *First*, we showed that our model compiler is able to generate optimized branchless byte code that can be executed as a fixed sequence of operations. Most importantly, this allows to compute the *required main- and GPU memory size* for a CalcGraph model and a given batch size—or an optimal batch size for a given memory size—*at compile time.*
– *Secondly*, we discussed how we minimize expensive memory allocations by preallocating the entire needed memory (computed by the model compiler) and then reusing it by switching models from storage to the preallocated memory and vice versa. This avoids expensive *malloc* and *free* operations

during runtime and allows us to run common model exploration tasks, such as hyperparameter optimization or neural architecture search, with constant memory. We refer to this as *model switching.*
- *Last but not least*, we discussed how knowing the memory requirements at compile time enables us to further *optimize the usage of the available resources* by scheduling as many models as possible for *parallel execution* and sharding the executions among the available resources.

We argued that these points are key enablers to tame the high costs of deep learning and allow for model exploration even on constrained hardware resources. As discussed, precomputing the required memory before the actual execution also avoids that a learning algorithm runs for hours or even days and then ends with an out of memory exception, a common problem in many frameworks today. CalcGraph is designed with model exploration in mind. We demonstrated the efficiency of CalcGraph by showing that it can outperform state-of-the-art frameworks like TensorFlow and PyTorch in terms of execution time as well as memory consumption. We evaluated a meta-learning scenario as a use-case of model exploration and identified the area (in terms of computational complexity), for which our approach yields the greatest advantages. We argue that this area is the most relevant for our target domain of flexible machine learning in real world applications with constrained hardware resources.

References

1. Deep Learning est mort. Vive Differentiable Programming?, 2018. [Online]. Available: `https://www.facebook.com/yann.lecun/posts/10155003011462143`, [Accessed: 02-August-2021]
2. Mimalloc memory allocator. [Online].
Available: `https://github.com/microsoft/mimalloc`, [Accessed: 02-August-2021]
3. ONNX Operators. [Online].
Available: `https://github.com/onnx/onnx/blob/master/docs/Operators.md`, [Accessed: 02-August-2021]
4. Optimising Lua. [Online].
Available: `http://www.mcours.net/cours/pdf/info/Cours_pdf_Optimising_Lua.pdf`, [Accessed: 02-August-2021]
5. TensorFlow Operators. [Online].
Available: `https://gist.github.com/dustinvtran/cf34557fb9388da4c9442ae25c2373c9`, [Accessed: 02-August-2021]
6. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, p. 265–283. USENIX Association, USA (2016)
7. Andrej Karpathy: Software 2.0. [Online]. Available: `https://medium.com/@karpathy/software-2-0-a64152b37c35`, [Accessed: 02-August-2021]
8. Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M.W., Pfau, D., Schaul, T., Shillingford, B., De Freitas, N.: Learning to learn by gradient descent by gradient descent. In: Advances in neural information processing systems, pp. 3981–3989 (2016)
9. Baydin, A.G., Pearlmutter, B.A., Radul, A.A., Siskind, J.M.: Automatic differentiation in machine learning: A survey. J. Mach. Learn. Res. **18**(1), 5595–5637 (2017)

10. Bengio, Y.: Learning deep architectures for ai. Found. Trends Mach. Learn. **2**(1), 1–127 (2009). DOI 10.1561/2200000006. URL `https://doi.org/10.1561/2200000006`
11. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. J. Mach. Learn. Res. **13**, 281–305 (2012)
12. Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, Berlin, Heidelberg (2006)
13. Blalock, D., Ortiz, J.J.G., Frankle, J., Guttag, J.: What is the State of Neural Network Pruning? arXiv:2003.03033 [cs, stat] (2020)
14. Bovet, D.P., Estrin, G.: On static memory allocation in computer systems. IEEE Transactions on Computers **100**(6), 492–503 (1970)
15. Byoung-Dai Lee, Schopf: Run-time prediction of parallel applications on shared environments. In: 2003 Proceedings IEEE International Conference on Cluster Computing, pp. 487–491 (2003)
16. Cabot, J., Clarisó, R., Brambilla, M., Gérard, S.: Cognifying model-driven software engineering. In: M. Seidl, S. Zschaler (eds.) STAF Workshops, *Lecture Notes in Computer Science*, vol. 10748, pp. 154–160. Springer (2017). URL `http://dblp.uni-trier.de/db/conf/staf/staf2017w.html#CabotCOG17`
17. Chen, C., Seff, A., Kornhauser, A., Xiao, J.: Deepdriving: Learning affordance for direct perception in autonomous driving. In: Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), ICCV '15, pp. 2722–2730. IEEE Computer Society, Washington, DC, USA (2015)
18. Chen, T., Xu, B., Zhang, C., Guestrin, C.: Training deep nets with sublinear memory cost (2016)
19. Cho, J.H., Hariharan, B.: On the Efficacy of Knowledge Distillation. roceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) p. 9 (2019)
20. Desell, T.: Large scale evolution of convolutional neural networks using volunteer computing. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '17, pp. 127–128. ACM, New York, NY, USA (2017)
21. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
22. Durner, D., Leis, V., Neumann, T.: On the impact of memory allocation on high-performance query processing. In: Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN'19. Association for Computing Machinery, New York, NY, USA (2019). DOI 10.1145/3329785.3329918. URL `https://doi.org/10.1145/3329785.3329918`
23. Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M.W., Keutzer, K.: A survey of quantization methods for efficient neural network inference (2021)
24. Hansen, L.K., Salamon, P.: Neural network ensembles. IEEE transactions on pattern analysis and machine intelligence **12**(10), 993–1001 (1990)
25. Hartmann, T., Moawad, A., Fouquet, F., Le Traon, Y.: The next evolution of mde: A seamless integration of machine learning into domain modeling. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 180–180 (2017)
26. Hartmann, T., Moawad, A., Fouquet, F., Le Traon, Y.: The next evolution of mde: A seamless integration of machine learning into domain modeling. Softw. Syst. Model. **18**(2), 1285–1304 (2019). DOI 10.1007/s10270-017-0600-2. URL `https://doi.org/10.1007/s10270-017-0600-2`
27. Hartmann, T., Moawad, A., Schockaert, C., Fouquet, F., Le Traon, Y.: Meta-modelling meta-learning. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 300–305 (2019)
28. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016). DOI 10.1109/CVPR.2016.90
29. He, K., Zhang, X., Ren, S., Sun, J.: Deep Residual Learning for Image Recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778. IEEE, Las Vegas, NV, USA (2016). DOI 10.1109/CVPR.2016.90
30. Hinton, G., Vinyals, O., Dean, J.: Distilling the Knowledge in a Neural Network. arXiv:1503.02531 [cs, stat] (2015)

31. Hu, Y., Anderson, L., Li, T.M., Sun, Q., Carr, N., Ragan-Kelley, J., Durand, F.: Difftaichi: Differentiable programming for physical simulation (2019)
32. Huang, G., Li, Y., Pleiss, G., Liu, Z., Hopcroft, J.E., Weinberger, K.Q.: Snapshot ensembles: Train 1, get m for free. arXiv preprint arXiv:1704.00109 (2017)
33. Innes, M., Edelman, A., Fischer, K., Rackauckas, C., Saba, E., Shah, V.B., Tebbutt, W.: A differentiable programming system to bridge machine learning and scientific computing (2019)
34. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., Kalenichenko, D.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2018)
35. Kim, T., Yoon, J., Dia, O., Kim, S., Bengio, Y., Ahn, S.: Bayesian model-agnostic meta-learning. CoRR **abs/1806.03836** (2018). URL `http://arxiv.org/abs/1806.03836`
36. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization (2014)
37. Kononenko, I.: Machine learning for medical diagnosis: History, state of the art and perspective. Artificial intelligence in medicine **23**, 89–109 (2001). DOI 10.1016/S0933-3657(01)00077-X
38. LeCun, Y., Denker, J.S., Solla, S.A.: Optimal Brain Damage. Advances in neural information processing systems p. 8 (1990)
39. Lee, S., Johnson, T., Raman, E.: Feedback directed optimization of tcmalloc. In: Proceedings of the Workshop on Memory Systems Performance and Correctness, MSPC '14. Association for Computing Machinery, New York, NY, USA (2014). DOI 10.1145/2618128.2618131. URL `https://doi.org/10.1145/2618128.2618131`
40. Li, H., Ota, K., Dong, M.: Learning iot in edge: Deep learning for the internet of things with edge computing. IEEE network **32**(1), 96–101 (2018)
41. Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: A novel bandit-based approach to hyperparameter optimization. J. Mach. Learn. Res. **18**(1), 6765–6816 (2017)
42. Liang, J., Meyerson, E., Hodjat, B., Fink, D., Mutch, K., Miikkulainen, R.: Evolutionary Neural AutoML for Deep Learning. arXiv e-prints arXiv:1902.06827 (2019)
43. Margossian, C.C.: A review of automatic differentiation and its efficient implementation. WIREs Data Mining and Knowledge Discovery **9**(4) (2019). DOI 10.1002/widm.1305. URL `http://dx.doi.org/10.1002/WIDM.1305`
44. Murshed, M.G.S., Murphy, C., Hou, D., Khan, N., Ananthanarayanan, G., Hussain, F.: Machine Learning at the Network Edge: A Survey. arXiv:1908.00080 [cs, stat] (2021)
45. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett (eds.) Advances in Neural Information Processing Systems 32, pp. 8026–8037. Curran Associates, Inc. (2019). URL `http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`
46. Pearlmutter, B.A., Siskind, J.M.: Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. ACM Trans. Program. Lang. Syst. **30**(2) (2008). DOI 10.1145/1330017.1330018. URL `https://doi.org/10.1145/1330017.1330018`
47. Plastiras, G., Terzi, M., Kyrkou, C., Theocharidcs, T.: Edge intelligence: Challenges and opportunities of near-sensor machine learning applications. In: 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 1–7 (2018). DOI 10.1109/ASAP.2018.8445118
48. Priya, R., de Souza, B.F., Rossi, A.L.D., de Carvalho, A.C.P.L.F.: Predicting execution time of machine learning tasks using metalearning. In: 2011 World Congress on Information and Communication Technologies, pp. 1193–1198 (2011)
49. Qin, Z., Yu, F., Liu, C., Chen, X.: How convolutional neural network see the world - A survey of convolutional neural network visualization methods. arXiv:1804.11191 [cs] (2018)

50. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision **115**(3), 211–252 (2015). DOI 10.1007/s11263-015-0816-y
51. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: MobileNetV2: Inverted Residuals and Linear Bottlenecks. arXiv:1801.04381 [cs] (2019)
52. Selvaraju, R.R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., Batra, D.: Grad-CAM: Visual Explanations From Deep Networks via Gradient-Based Localization. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 618–626 (2017)
53. Shrikumar, A., Greenside, P., Kundaje, A.: Learning important features through propagating activation differences. In: International Conference on Machine Learning, pp. 3145–3153. PMLR (2017)
54. Simonyan, K., Vedaldi, A., Zisserman, A.: Deep inside convolutional networks: Visualising image classification models and saliency maps. arXiv preprint arXiv:1312.6034 (2013)
55. Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. Advances in neural information processing systems **25** (2012)
56. Stojanovic, R., Mitropulos, P., Koulamas, C., Karayiannis, Y., Koubias, S., Papadopoulos, G.: Real-time vision-based system for textile fabric inspection. Real-Time Imaging **7**(6), 507–518 (2001)
57. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, NIPS'14, pp. 3104–3112. MIT Press, Cambridge, MA, USA (2014)
58. Süzen, A.A., Duman, B., Şen, B.: Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn. In: 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), pp. 1–5. IEEE (2020)
59. Szegedy, C., Wei Liu, Yangqing Jia, Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1–9. IEEE, Boston, MA, USA (2015). DOI 10.1109/CVPR.2015.7298594
60. Vanschoren, J.: Meta-learning: A survey. arXiv preprint arXiv:1810.03548 (2018)
61. Vilalta, R., Drissi, Y.: A perspective view and survey of meta-learning. Artif. Intell. Rev. **18**(2), 77–95 (2002). DOI 10.1023/A:1019956318069. URL `https://doi.org/10.1023/A:1019956318069`
62. Wang, F., Decker, J., Wu, X., Essertel, G., Rompf, T.: Backpropagation with continuation callbacks: Foundations for efficient and expressive differentiable programming. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, p. 10201–10212. Curran Associates Inc., Red Hook, NY, USA (2018)
63. Wei, R., Zheng, D., Rasi, M., Chrzaszcz, B.: Differentiable Programming Manifesto. [Online]. Available: `https://github.com/apple/swift/blob/master/docs/DifferentiableProgramming.md`, [Accessed: 02-August-2021]
64. Wenzel, F., Snoek, J., Tran, D., Jenatton, R.: Hyperparameter ensembles for robustness and uncertainty quantification (2021)
65. Wimmer, C., Würthinger, T.: Truffle: A self-optimizing runtime system. In: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, p. 13–14. Association for Computing Machinery, New York, NY, USA (2012). DOI 10.1145/2384716.2384723. URL `https://doi.org/10.1145/2384716.2384723`
66. Zhang, S., Choromanska, A., LeCun, Y.: Deep learning with elastic averaging sgd (2015)
67. Zhang, Y., Tang, H., Jia, K.: Fine-grained visual categorization using meta-learning optimization with sample selection of auxiliary data (2018)
68. Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., Torralba, A.: Learning deep features for discriminative localization. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 2921–2929 (2016)
69. Zoph, B., Le, Q.V.: Neural Architecture Search with Reinforcement Learning. arXiv e-prints arXiv:1611.01578 (2016)
70. Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 8697–8710 (2018)