Amanda: Unified Instrumentation Framework for Deep Neural Networks

Yue Guan[*]
bonboru@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute

Yuxian Qiu[*]
qiuyuxian@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute

Jingwen Leng[†]
leng-jw@cs.sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute

Fan Yang
fanyang@microsoft.com
Microsoft Research

Shuo Yu
y-shuo@sjtu.edu.cn
Shanghai Jiao Tong University

Yunxin Liu
liuyunxin@air.tsinghua.edu.cn
Tsinghua University
Shanghai Artificial Intelligence Laboratory

Yu Feng
yfeng28@ur.rochester.edu
University of Rochester

Yuhao Zhu
yzhu@rochester.edu
University of Rochester

Lidong Zhou
lidongz@microsoft.com
Microsoft Research

Yun Liang
ericlyun@pku.edu.cn
Peking University

Chen Zhang
chenzhang.sjtu@sjtu.edu.cn
Shanghai Jiao Tong University

Chao Li
lichao@cs.sjtu.edu.cn
Shanghai Jiao Tong University

Minyi Guo[†]
guo-my@cs.sjtu.edu.cn
Shanghai Jiao Tong University

Abstract

The success of deep neural networks (DNNs) has sparked efforts to analyze (e.g., tracing) and optimize (e.g., pruning) them. These tasks have specific requirements and ad-hoc implementations in current execution backends like TensorFlow/PyTorch, which require developers to manage fragmented interfaces and adapt their codes to diverse models. In this study, we propose a new framework called Amanda to streamline the development of these tasks. We formalize the implementation of these tasks as **neural network instrumentation**, which involves introducing instrumentation into the operator level of DNNs. This allows us to abstract DNN analysis and optimization tasks as instrumentation tools on various DNN models. We build Amanda with two levels of APIs to achieve a unified, extensible, and efficient instrumentation design. The user-level API provides a unified operator-grained instrumentation API for different backends. Meanwhile, internally, we design a set of callback-centric APIs for managing and optimizing the execution of original and instrumentation codes in different backends. Through these design principles, the Amanda framework can accommodate a broad spectrum of use cases, such as tracing, profiling, pruning, and quantization, across different backends (e.g., TensorFlow/PyTorch) and execution modes (graph/eager mode). Moreover, our efficient execution management ensures that the performance overhead is typically kept within 5%.

[*]Both authors contributed equally to this research.
[†]Jingwen Leng and Minyi Guo are corresponding authors of this paper.

Conference, April 27–May 1,2024, San Diego, CA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Yue Guan, Yuxian Qiu, Jingwen Leng, Fan Yang, Shuo Yu, Yunxin Liu, Yu Feng, Yuhao Zhu, Lidong Zhou, Yun Liang, Chen Zhang, Chao Li, and Minyi Guo. 2023. Amanda: Unified Instrumentation Framework for Deep Neural Networks. In *Proceedings of ASPLOS (Conference).* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Deep neural network (DNN) has emerged as the dominant technology in various intelligent tasks such as natural language understanding [54, 73], computer vision [29, 50, 74], and speech recognition [84]. However, DNNs are difficult to deploy owing to their excessive computing and memory consumption [23, 34] as well as black-box nature [28, 53]. Therefore, numerous research works focus on the analysis and optimization of DNNs. Examples include tracing the activation of neurons revealing new insights into DNN inferences [37, 63], optimizing DNN memory usage [69, 75], and compressing DNNs through quantization or pruning [40, 41, 43, 47].

The common requirements of the DNN analysis and optimization tasks are to **monitor** and **manipulate** the execution process of an existing DNN model. However, we find that researchers or developers implement those tasks in an ad-hoc way because of the limited support from execution backends, including TensorFlow/PyTorch. Different tasks require a diverse set of modification points and computation states (e.g., various tensors). For example, weight pruning [19] improves DNNs' computation efficiency by compressing their static weight tensors *at every training iteration*, while activation pruning [78, 80] and quantization [35] are performed *at each layer*. Quantization tasks [35] may also require activation tensors and corresponding gradient tensors in the back-propagation process. As a result, different tasks rely on fragmented approaches, such as PyTorch's module hook [68], TensorFlow's session hook [17], and even source-code wrapping.

Due to the inconsistent DNN modification points and computational states, developers must manually deal with the wide interaction surface between their analysis/optimization tasks and DNN models. This manual process may lead to implementations that are bound to a particular execution backend or even a specific model. Moreover, developers with a machine learning background may need to understand the low-level details of the underlying execution backends, such as the **graph and eager execution mode**. The graph mode separates the model building and the execution stage, while the eager mode evaluates the user operations immediately [67]. The graph mode allows for the global graph structure of the model to be available, but modifications require the model to be rewritten. And the eager mode does not have an explicit graph structure but can introduce complex control flow operations. This additional level of inconsistency presents significant challenges for researchers in their efforts to analyze and optimize DNNs.

In this study, we propose addressing the aforementioned fragmented situation by borrowing the mature instrumentation concept from program analysis [61, 79]. We extend the instrumentation concept to DNNs' operators, which are multi-input/-output functions with high-level model semantics. With our approach, users can abstract the analysis and optimization tasks as instrumentation tools and implement those tools by inserting codes to monitor or manipulate the execution process of a target DNN without interacting with its implementation detail. Our instrumentation model unifies the diverse modification points to a universal operator instrumentation point abstraction, which we find is the finest granularity required in those tasks. Other instrumentation points, such as iteration, forward, and backward processes can be derived by combining the operator-level instrumentation point and context.

With the above insights, we present Amanda framework, a **unified, portable, reusable, and composable** instrumentation infrastructure to facilitate the analysis and optimization tasks on existing DNN models. Amanda provides the *unified* operator-centric user-level instrumentation APIs that hide the details of underlying execution backends. These APIs cover a variety of instrumentation points and provide access to rich instrumentation context with different tensors in the forward and backward computation. They let developers monitor and manipulate DNNs without concern about the complexity of accessing their implementation details.

As a result, instrumentation tools implemented through Amanda are decoupled from current ad-hoc instrumentation practices, making them DNN-model-independent and *portable* to different DNNs in the same backend framework. Furthermore, Amanda also provides an extra context mapping instrumentation tool that transforms the backend framework's context in advance, making those instrumentation tools even more portable across backend frameworks.

We take a layered approach to make Amanda tools *reusable* and *composable*. To accommodate common requirements across different tasks, Amanda allows users to specify dependencies of other instrumentation tools and internally applies multiple tools. In this sense, Amanda tools themselves are layered, where low-layer tools can be reused and composed into a higher-layer tool that facilitates instrumentation at the higher layer. Current practice makes such composition difficult, if not impossible. We envision that experienced Amanda users would contribute their individual tools in the future, making Amanda a community-maintained and long-term-evolving general infrastructure. Moreover, Amanda encloses several built-in instrumentation tools to address design challenges, including the one that builds a computation graph in the eager mode, which users can pre-load to access the graph-level structure in the instrumentation context.

Our work achieves the above unified user-level APIs across different backends (i.e., TensorFlow/PyTorch) and different execution modes (i.e., graph/eager) by adopting a two-layer modular design. We first build a backend-dependent layer called backend driver that handles the per-backend complexity. These drivers offer common low-level callback-centric APIs that implement new or utilize existing callback mechanisms for each backend. We then construct our universal operator-level instrumentation model on top of this common callback mechanism. Currently, we have implemented drivers for PyTorch and TensorFlow, respectively, for the eager and graph execution mode in Amanda.

We also construct a backend-independent layer that is called Amanda-core on top of the common callback-centric backend interface. The aforementioned backend interface defines the required instrumentation points to register callbacks and contexts from the execution. This layer manages callbacks and maps them with operator granularity, even caching them to minimize instrumentation overhead. The design makes Amanda easily extendable to different DNN execution backends with a clearly defined backend interface.

To elaborate, we have extensively evaluated Amanda with multiple neural network instrumentation tasks, including tracing, profiling, pruning, and quantization. We show that these tools can be implemented with Amanda's APIs with fewer lines of code than a direct implementation. The overhead of Amanda is less than 5% for most cases, making it applicable in real-world scenarios.

We make the following contributions in this work.

- We propose a novel DNN instrumentation concept with simple operator abstraction to support analysis and optimization applications of DNN models.
- We propose the Amanda DNN instrumentation framework to support DNN instrumentation applications and implement the framework on two mainstream backends.
- We develop multiple instrumentation tools for representative DNN analysis and optimization tasks using Amanda to evaluate its generality, consistency, and feasibility.

2 Background and Motivation

The analysis and optimization of DNNs. With DNNs' recent success, a large number of works have emerged, which aim to analyze [12, 89] or accelerate [76] their inference or training. The common requirement of these tasks is to monitor and manipulate the computation process of DNN models, whose behavior closely resembles the general concept of instrumentation in traditional program analysis. To better stress how the concept of instrumentation supports those analysis and optimization tasks, we refer to them as **DNN instrumentation tasks** and use abstraction and terminology of instrumentation to analyze those tasks. In this section, we show that current practices for DNN instrumentation tasks are far less mature and even fragmented compared with program instrumentations. Several major difficulties have to be conquered by the researchers or developers to facilitate a specific task.

The pain of ad-hoc instrumentation points. When instrumenting a DNN model, applications require a set of diverse instrumentation points. That is where the runtime state information is accessed or the manipulation operation is applied. With current DNN execution backends, there exist many different interfaces to access the DNN. As such, ad hoc implementation designs are required accordingly.

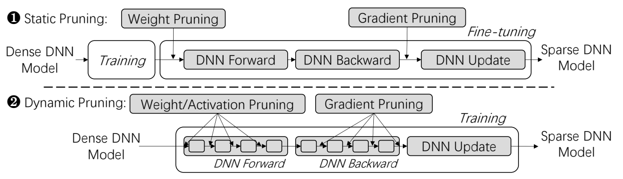

Fig. 1. Instrumentation points of different pruning methods.

We use network pruning algorithms, a popular approach to compress DNN models for better computation efficiency, as examples here. The pruning process exploits the inherent redundancy in the DNN models to transform the original, dense model into a sparse model with fewer parameters and lower computation costs. There are vast works of literature on DNN pruning [27, 48, 58, 60, 62, 72, 81], which can be generally divided into static and dynamic pruning methods. The static pruning method [48] only removes the model weight parameters statically and performs finetuning with extra training iterations to recover the accuracy loss. Different static pruning methods may apply different sparsity patterns such as the unstructured element-wise pattern [45] and structured channel-wise pattern [52], which lead to different accuracy and computation efficiency trade-off [40]. PyTorch and Tensorflow provide built-in tools for supporting the basic static pruning method [9, 13]. However, they only support a few fixed sparsity patterns, and their interfaces are counter-intuitive or even unusable [40]. As such, researchers often need to manually wrap the step function of the optimizer to implement their own static pruning methods. The example of third-party pruning tool APEX [7] in Fig. 1❶ takes such an optimizer-wrapping approach, which manually updates weight tensors and their associated gradients in the fine-tuning process.

The above implementation relies on the container object `module` in PyTorch to access the parameters statically, which cannot work for other dynamic pruning methods with more complex instrumentation points requirements. The dynamic pruning method determines sparsity during training and updates it during the procedure, which generally has better accuracy, however, more complicated pruning process [38, 39, 49]. Given its dynamic nature, these methods prune both weight tensors and activation tensors. As such, they require operator-level instrumentation points to access the runtime activation tensors and the corresponding gradients as shown with Fig. 1❷. To insert cuszomized operators or algorithms at these instrumentation points, developers have to deal with ad-hoc interfaces and programming models.

The pain of fragmented state representations. Various DNN instrumentation tasks also demand ad-hoc state representations. We use another widely used DNN compression method quantization [35, 43, 44, 86], which reduces

	Weight	Weight Gradient	Activation	Activation Gradient	Instrumentation Points	Graph
Quantization Methods						
Static PTQ [55]	✓	✗	✗	✗	Operator	✗
Dynamic PTQ [83]	✓	✗	✓	✗	Operator	✗
QAT [66]	✓	✓	✓	✓	Operator	✗
Other Instrumentation Tasks						
Weight Pruning [48]	✓	✓	✗	✗	Iteration	✗
Activation Pruning [78]	✗	✗	✓	✓	Operator	✗
Profiling [15]	✓	✗	✓	✗	Operator	✗
Effective Path [70]	✓	✓	✓	✓	Operator	✓
DTR [57]	✓	✗	✓	✗	Operator	✓
Instrumentation Interfaces in Current Execution Backends						
Source Modification	✓	✓	✓	✗	Operator	✗
Module Hook	Partial	Partial	Partial	Partial	Module	✗
Amanda	✓	✓	✓	✓	Operator	✓

Table 1. The computation state requirements of different DNN model optimization/analysis tasks (top two parts) and capabilities of current frameworks (bottom part).

the number of bits for DNN tensors, as an example to illustrate the point. Quantizing 32-bit floating-point-based DNNs with 8-bit integer numbers achieves 4× storage and memory bandwidth reduction and also simplifies the hardware processing units. According to their usages, quantization methods are categorized into static post-training quantization (static PTQ) [55], dynamic post-training quantization (dynamic PTQ) [83] and quantization aware training (QAT) [66].

These above quantization methods require different computation states, which we summarize in Tbl. 1. First, they require different types of tensors. The static PTQ only compresses the weight tensors, while dynamic PTQ compresses both weight and activation tensors. The QAT uses extra training steps to mitigate the accuracy loss caused by quantization, which further requires gradient tensors.

We find that the diverse requirements of quantization methods lead to fragmented implementations. The current built-in QAT tool of PyTorch framework only supports basic convolution and linear operators [14]. To implement their own quantization methods, developers often manually wrap every PyTorch module with additional quantization operators. However, many DNN models also use PyTorch's functional APIs [11], for which the above module-wrapping method fails. As such, massive ad-hoc efforts are needed to extract the desired state representations from each DNN model.

The extra complexity dimension of execution modes. Together with the aforementioned ad-hoc instrumentation points and representation states, the backend execution modes, which include equally popular graph and eager mode, add another dimension of complexity for users to implement their instrumentation tasks. In the graph mode, each model is first compiled into a computation graph consisting of operators and then feed to runtime for execution. In contrast, operators are executed immediately in eager mode. Those two modes provide different interfaces for implementing DNN instrumentation tasks, as shown in the last part of Tbl. 1.

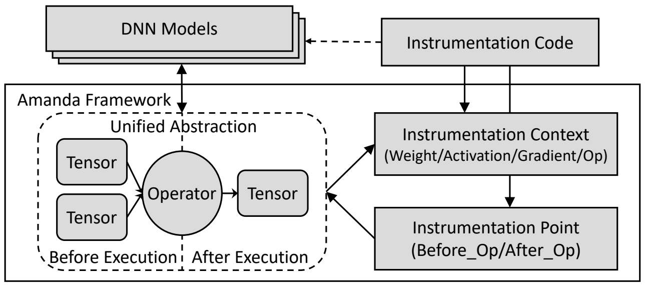

Fig. 2. The operator-based instrumentation abstraction adopted by Amanda to support DNN optimization tasks.

As such, developers have to deal with inconsistency from not only the ad-hoc instrumentation points and fragmented state representations, and also the complexity introduced by the execution backend.

We use the sparsity profiling workload [39] in Tbl. 1 to highlight its different implementation rationals in different backends. This workload examines the sparsity of weight/activation tensors in a DNN model. In the eager mode such as PyTorch, users can easily modify the source code, while users need to use the callback mechanism in the graph mode of TensorFlow to achieve the same functionality [67].

Summary. Tbl. 1 summarizes the required instrumentation points and computational states in various studied DNN instrumentation tasks (first two parts). Besides various tensors, certain tasks such as effective path [70] and dynamic tensor rematerialization [57] also require the model's graph structure: the former uses it to extract sparse activation for interpreting the model's inference process while the latter use it for memory management optimization. Current backends have many inconveniences and limitations to support those tasks (last part). This motivates us to design and implement Amanda, which provides a universal abstraction for DNN instrumentation with a wide range of instrumentation points and state representations. Amanda facilitates the development of portable DNN instrumentation tasks.

3 Operator Instrumentation Abstraction

To address the fragmented choices of current DNN optimization and analysis tasks, we propose to bring the wisdom of program instrumentation into DNN frameworks, including PyTorch and TensorFlow. To the best of our knowledge, we are the first to formally introduce such DNN level instrumentation and provide principled support. We explain the similarities and differences between DNN and the program.

Program instrumentation frameworks, such as Pin [61] and NVbit [79], have been widely used for various purposes, such as performance analysis, error diagnosis and information tracing. These frameworks provide user-friendly APIs that allow developers to insert analysis or emulation codes into a target program without modifying the source program.

The same instrumentation codes can be reused for different programs, making instrumentation more efficient.

Similarly, DNN-level instrumentation involves inserting user-defined codes or operators into a target DNN model executed on a certain backend like TensorFlow and PyTorch. With this programming abstraction, developers can write codes for their tasks directly without touching the DNN model source code, as shown in Fig. 2. Users write instrumentation code given the computation states as instrumentation context and specify the desired instrumentation points. This decouples the implementation of optimization and analysis tasks from the implementation of DNN models.

Different from the program instrumentation, we advocate the operator-level instrumentation abstraction for DNN instrumentation. Our key observation is that modern auto-differentiation frameworks for DNNs are all built with the same operator-based design [1, 17, 21, 68]. This design characterizes a DNN model as a data-flow graph (DFG) of operators and perform auto-differentiation for each operators reversely. The fundamental abstraction is the basic execution functions as nodes in the neural network DFG with multiple tensor or scalar inputs and outputs. Although different AD frameworks consists of diverse operator sets and implementations, this abstraction is always available. Therefore, we can inspect and modify the execution of DNNs with a unified operator instrumentation interface, similar to instruction-based binary instrumentation. Tbl. 2 shows the conceptual comparison of binary and DNN instrumentation.

Our operator-based instrumentation abstraction, shown in Fig. 2 provides simple callback trigger points before and after the execution of an operator. Under this programming abstraction, the ad-hoc DNN model entry points for different tasks are unified as instrumentation points surrounding the operators. Higher-level instrumentation points, such as modules or subgraphs, can be composed of operator-level instrumentation points by locating the proper operator to insert the higher-level instrumentation codes.

Futhermore, the operator-based abstraction lets users access different tensors as edges connecting two operators in the instrumentation context. As all the computation state tensors are connected to the computation graph as edges, diverse state representations are accessible through the same interface, including dynamic activation tensors and implicit backward gradient tensors. The abstraction of operators and

	Instrumentation Points			State Representations
Binary	Instruction	Function	Program	Register/Memory/ Type/...
DNN	Operator	Module/ Subgraph	Graph	Weight/Activation/ Gradient/...

Table 2. Binary and DNN instrumentation comparison.

```python
class PruningTool(amanda.Tool):
  def __init__(self):
    self.depends_on(
      MappingTool(rules=[["tensorflow", tf_type],...,]))
    # register callbacks in forward and backward execution
    self.add_inst_for_op(self.instrumentation)
    self.add_inst_for_op(self.backward_instrumentation,
                         backward=True,
                         require_outputs=True)
  # arbitrary pruning algorithm
  def get_mask(self, tensor: Tensor) -> Tensor:
    ...
  # analysis routines
  def instrumentation(self, context: amanda.OpContext):
    if context["type"] in ["conv2d", ]:
      weight = context.get_inputs()[1]
      mask = self.get_mask(weight)
      context["mask"] = mask
      context.insert_before_op(self.mask_forward_weight,
                               inputs=[1], mask=mask)
  def backward_instrumentation(self, context: amanda.OpContext):
    if context["backward_type"] in ["conv2d_backward",]:
      weight_grad = context.get_grad_inputs()[0]
      mask = context["mask"]
      context.insert_after_backward_op(
        self.mask_backward_gradient, grad_inputs=[0], mask=mask)
  # instrumentation routines
  def mask_forward_weight(self, weight, mask):
    return weight * mask
  def mask_backward_gradient(self, weight_grad, mask):
    return weight_grad * mask
# apply instrumentation tool to DNN execution
with amanda.apply(PruningTool()):
  resnet50(model_input)
```

Listing 1. Amanda's operator instrumentation tool example with a pruning tool on PyTorch backend.

tensors is independent of the underlying execution backends. As such, users can insert codes at their specified points with access to a rich instrumentation context, without knowing or spending the efforts to modify the implementation details of the underlying neural network model.

4 Amanda Instrumentation Interface

Following the previously discussed abstraction, we present the user-level interface of the Amanda instrumentation framework. Next, a minimum example is provided to demonstrate the usage of Amanda and its instrumentation tools.

4.1 User Level Interface Introduction

The Amanda framework provides a user interface that is based on general instrumentation interfaces borrowed from program instrumentation frameworks. Users can use those interfaces to implement their DNN analysis and optimization tasks as Amanda tools composed of analysis routines and instrumentations routines. The analysis routines analyze the program or DNN model statically without runtime states. The analysis routines filter and locate the desired position

```
class Tool:
  def add_inst_for_op(
    self,
    callback: Callable[[OpContext], None],
    backward: bool = False,
    require_outputs: bool = False,
  ) -> None:
  def depends_on(self, *tools: Tool) -> None:
```

Listing 2. Registration APIs: register analysis routines.

```
class OpContext(dict):
  insert_before_op(self, func, inputs, **kwargs)
  insert_after_op(self, func, outputs, **kwargs)
  insert_before_backward_op(self,func,grad_out,**
       kwargs)
  insert_after_backward_op(self,func,grad_in,**
       kwargs)
  replace_op(self, func, inputs, **kwargs)
  replace_backward_op(self,func,grad_out,**kwargs)
```

Listing 3. Instrumentation APIs: modify models.

to insert instrumentation routines defined by the same tool. These instrumentation routines, which are extra codes or operators inserted into the original DNN execution process, let users collect data and perform various optimization tasks.

The user-level interfaces in Amanda are provided as a Python library with C++ extensions. To implement a DNN optimization or analysis task as an Amanda tool, users can declare a class that inherits from `amanda.Tool`, or create an `amanda.Tool` instance directly. The class-based method provides a language-native way to manage the tool's state through instance variables, while the alternative method is more suitable for stateless tools or one-off use cases.

Example. As a concrete example, we walk through Lst. 1 to demonstrate how to these concepts to implement an instrumentation tool that can locate and prune the weight tensor of convolution operators in any DNN models [40]. To begin, we register analysis routines (Line 6 & 7) that determine the forward and backward convolution operators to prune. Within the analysis routine, Amanda provides the context of each operator. In the forward analysis routine of this instance, we retrieve the operator type from the context (Line 14). If the operator is a *conv2d*, we compute its pruning mask using an arbitrary `get_mask()` method, which is determined by the pruning algorithms (Line 16). Then we insert a forward instrumentation routine just before the operator executes pruning with the sparse mask at runtime (Line 18). This instrumentation routine prunes the weight tensor each time the operator is executed (Line 27). We also inject the mask into the instrumentation context so that the backward gradient pruning can access the same mask (Line 17). Backward instrumentation pruning the gradient of convolution weights is similar to the forward and the analysis and instrumentation routines are defined in Lines 20 and 29. This tool can be easily applied to any DNN model without the need to reference its source code (Line 32).

```
class OpContext(dict):
  def get_op(self):
  def get_op_id(self):
  def get_inputs(self):
  def get_outputs(self):
  def get_backward_op(self):
  def get_backward_op_id(self):
  def get_grad_outputs(self):
  def get_grad_inputs(self):
```

Listing 4. Inspection APIs: retrieve op information.

4.2 Amanda Tool APIs

Tool APIs are the key user-level APIs Amanda provided and called by the instrumentation tool instances. They are categorized into four groups as registration APIs, instrumentation APIs, inspection APIs and control APIs.

Registration APIs are used to register callbacks for all forward or backward ops, which are referred to as analysis routines. The registration APIs are shown in Lst. 2. There are two arguments specifying the exact instrumentation point of the analysis routine. They are `backward` specifying the backward graph and `require_outputs` specifying after the execution of the operator. The combination of the two arguments leads to four different registration locations, which are before the forward operator (`before_forward_op`), after the forward operator (`after_forward_op`), before the backward operator (`before_backward_op`), and after the backward operator (`after_backward_op`).

Instrumentation APIs modify the DNN model on-the-fly with the specified codes or operator. Thus it provide users the capability of accessing or modifying the instrumentation context at runtime, which is referred to as instrumentation routines. As shown in Lst. 3, modification can be achieved by injecting instrumentation routines before or after a target operator. Different from the analysis routines, instrumentation routines are called dynamically each time an operator executes with runtime states. For example, to perform some actions before an op is executed, users provide a function, which accepts the original input tensors of the op and returns the updated input tensors, to `insert_before_op()` API. Instrumentation APIs also have a parameter, such as `inputs` indicating which computation states are required by the instrumentation routine. Besides, Amanda allows instrumentation tools to inject outside parameters into the instrumentation routines with `kwargs`. That means the inserted operator can take additional states into the original DNN execution during execution. Amanda supports two kinds of modifications of the DNN operators: insertion and replacement. By replacing the operator with an identity operator which forwards its inputs, the removal semantic of operator can be also achieved.

Inspection APIs provide computation states as instrumentation context illustrated in Lst. 4. The available context for each operator includes the operator metadata, the

```python
def apply(*tools: Tool):
def disabled():
def enabled():
def cache_disabled():
def cache_enabled():
...
```

Listing 5. Control APIs: enable and disable tools or caches.

corresponding operator in the backward process, and the input/output tensors of these two operators. In addition, Amanda also assigns a unique label (or ID) for each operator, which is similar to the program counter value of each instruction in a program. Users can leverage this ID to aggregate statistical metrics across multiple iterations or store extra operator information. Note that those inspection APIs are called in the analysis routine but the context is provided for instrumentation routine at runtime.

Control APIs let users control Amanda's instrumentation behavior. Users can leverage the first two `disabled()` and `enabled()` APIs in Lst. 5 to specify the instrumentation scope in the DNN model. Instrumentation tools will be applied to all executed operators executed inside the `apply()` API unless Amanda is disabled on purpose. Similarly, the cache control APIs manages the scope of the cache mechanism of analysis routines, which we will explain later.

5 Amanda System Design

In this section, we first present the design overview of Amanda that implements the DNN instrumentation interface in a layered fashion to address the divergences of different hierarchies. We then demonstrate the challenges and solutions in Amanda called Amanda-core. Finally, we explain how Amanda bridges to different backends with modular drivers to make the instrumentation infrastructure extendable.

5.1 Amanda System Overview

The system design of Amanda is depicted in Fig. 3. Amanda comprises two layers of interfaces: the tool APIs and backend APIs, represented as different boxes. The former APIs interact with high-level user tools while the latter APIs interact with low-level DNN backends, respectively. As explained in the previous section, the user-level tool APIs are provided for users to develop instrumentation tools for various DNN analysis or optimization tasks. Additionally, certain features of Amanda's framework are delivered as built-in instrumentation tools, such as context transformation and graph structural tracing. These tools are also implemented using the same user-level instrumentation interface, demonstrating the feasibility and expressiveness of our instrumentation framework design, as elaborated later in this paper.

The Amanda system employs backend APIs to effectively separate the backend-independent Amanda-core from the backend-dependent drivers. As discussed in the Sec. 2, the current DNN instrumentation tasks exhibit divergences at

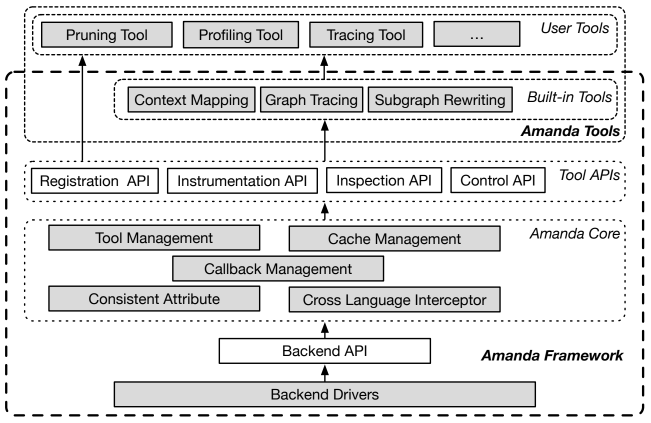

Fig. 3. Architecture of Amanda instrumentation framework.

multiple hierarchies. To address the complexity arising from different backends and to establish a unified interface, Amanda adopts a layered system design that segregates the backend-related and irrelevant components as Fig. 3 shows. The modular backend drivers implement the necessary raw callback mechanism for operator instrumentation at each backend's auto-differentiation execution engine. The Amanda core manages the inserted raw callbacks, implementing the operator instrumentation abstraction and optimizing for efficiency. To facilitate this, the backend API defines the required raw call back entry, instrumentation contexts, and meta information for their management.

This layered design of Amanda promotes transparency and adaptability with respect to the execution backends. To accommodate a new backend, we only need to implement the backend-dependent drivers and support the required backend APIs. In the following, we introduce the detailed implementation of Amanda-core and backend drivers.

5.2 Amanda Core

The Amanda-core is composed of multiple components as shown in Fig. 3. They are integrated to effectively tackle the design and implementation challenges faced by the system. We highlight and address several of these challenges.

Harmonizing instrumentation semantics on different execution modes. One of the challenges for DNN instrumentation is the presence of multiple execution modes in the backends, namely eager mode and graph mode. In the graph execution mode, analysis routines are statically executed during the construction of the computation graph, while instrumentation routines are dynamically inserted and processed in the DNN execution graph, as illustrated in Fig. 4 ❷. This implementation approach is similar to binary instrumentation methods [61, 79], where analysis routines are invoked during the recompilation of the binary program to its instrumented version, prior to actual execution.

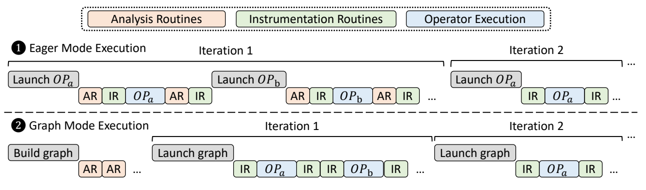

Fig. 4. Analysis routines and instrumentation routines with operator abstraction on eager and graph execution mode.

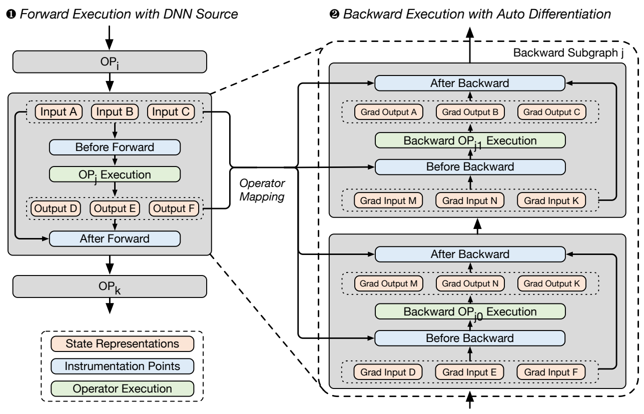

Fig. 5. Operator mapping of AD instrumentation.

However, the eager execution mode executes each operators instantly without an explicit graph-building phase, for which we propose the lazy instrumentation of analysis routines when the operator is executed for the first time as shown with ❶. The instrumentation routines are triggered eagerly after the analysis routine's registration. To elaborate, Amanda core assigns a consistent attribute ID for each operator with linear congruential generator (LCG) [46] to track their execution between iterations. The callback management triggers analysis routines and keeps the instrumentation routines' log for later iterations.

Addressing AD mechanism. Another challenge for DNN instrumentation is how to handle of auto-differentiation (AD) mechanism in existing DNN frameworks. With such a mechanism, users only need to declare the forward execution part of the DNN, and the framework automatically performs back-propagation [36], which is not exposed to end users. Unlike traditional program instrumentation, which only deals with the original program, DNN instrumentation needs to take into account both the original "program" (i.e., forward computation) and the AD program (i.e., backward computation).

The design of Amanda includes support for registering instrumentation routines to the backward operators within the forward analysis routine, achieved by specifying the *backward* argument in the registration API as shown in Fig. 5. This design feature simplifies analysis and optimization tasks such as fine-tuning after pruning [40], as developers do not need to collect the state representations across the two programs manually. Amanda is responsible for managing operator callbacks between the two programs to ensure the correct provision of state representations and instrumentation routines. Additionally, since one forward operator can launch multiple backward operators for gradient computation, Amanda-core tracks the backward operators launched by each forward operator and provides instrumentation points and forward context for all of them. This allows for the visibility of hidden backward state representations, such as the intermediate gradient tensor.

Another challenge brought by the auto-differentiation (AD) feature is that the instrumented code in the forward graph could alter the original backward graph at runtime. To address this issue, the Amanda-core disables the AD for inserted instrumentation routines to isolate the instrumented code. For instance, if an instrumentation tool intends to gather forward activation tensors and calculate their L1 norms, the inserted forward operators should not modify the backward process with L1 gradient calculations.

To provide flexibility for experienced users, we also provide a control interface in Lst. 5 to overwrite the above behavior, i.e., allowing the inserted operators to participate in the backward graph. This feature is useful when the instrumented code should participate in the backward calculation such as the fine-tuning after pruning [40]. The pruning tool might aim to insert sparsity masks for both the forward *matmul* operator and its corresponding backward *Matmul-Backward* operator. This can be achieved by enabling AD of instrumented code and only instrumenting the forward process. As this represents an advanced feature, the responsibility for ensuring the safety of AD in instrumented code lies with the tool developers.

Composable tools and context transformation. We observe that certain tasks may share similar requirements, such as sparsity profiling and standard context transformation in different pruning algorithms. To prevent redundant work and encourage the reuse of low-level tools, Amanda enables the usage of layered composable tools, which allows multiple instrumentation tools to be used, as shown in Fig. 6.

Users can utilize the `depends_on()` registration API in Lst. 2 to specify the dependencies for their tools. These dependencies determine the triggering order of instrumentation routines and the transformation of operator contexts. Specifically, an instrumentation tool is considered dependent on another tool when it needs to consume and be triggered after the other tool. Amanda-core handles tool management by resolving the dependency graph of instrumentation tools during initialization and detecting loop dependencies. During execution, the tools transform the instrumentation context according to the determined dependency graph.

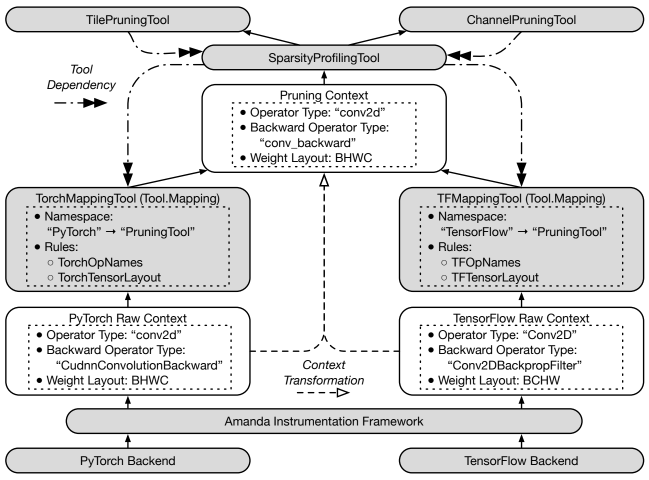
TilePruningTool
ChannelPruningTool
SparsityProfilingTool
Tool Dependency
Pruning Context
● Operator Type: "conv2d"
● Backward Operator Type: "conv_backward"
● Weight Layout: BHWC
TorchMappingTool (Tool.Mapping)
● Namespace: "PyTorch" → "PruningTool"
● Rules:
○ TorchOpNames
○ TorchTensorLayout
TFMappingTool (Tool.Mapping)
● Namespace: "TensorFlow" → "PruningTool"
● Rules:
○ TFOpNames
○ TFTensorLayout
PyTorch Raw Context
● Operator Type: "conv2d"
● Backward Operator Type: "CudnnConvolutionBackward"
● Weight Layout: BHWC
Context Transformation
TensorFlow Raw Context
● Operator Type: "Conv2D"
● Backward Operator Type: "Conv2DBackpropFilter"
● Weight Layout: BCHW
Amanda Instrumentation Framework
PyTorch Backend
TensorFlow Backend

Fig. 6. Composable instrumentation tools that transform the context, like name and tensor layout, to a common format.

```python
def tf_type(context: amanda.OpContext):
  op = context.get_op()
  context["type"] = op.type.lower()
  if not context.is_forward():
    backward_op = context.get_backward_op()
    if backward_op.type == "Conv2DBackpropFilter":
      context["backward_type"] = "conv2d_backward"
class PruningTool(amanda.Tool):
  def __init__(self):
    self.depends_on(
      amanda.tools.mapping.MappingTool(
        rules=[ ["tensorflow", tf_type],
                ["tensorflow", tf_get_shape],
                ["tensorflow", tf_get_mask],
                ["pytorch", torch_type],
                ["pytorch", torch_get_shape],
                ["pytorch", torch_get_mask], ]))
```

Listing 6. Context mapping tool example with transformation rules.

Certain design features of Amanda are supported as built-in tools that can be directly utilized by the user tool through pre-loading. For instance, we have designed and implemented the **graph tracing** tool that provides the graph structure of the DNN model in the instrumentation context, enabling tasks [42] that require a global view or the ability to look back from the current operator. Additionally, we have implemented another **subgraph rewriting** tool that allows for modifications on a subgraph granularity of the DNN model. We anticipate that in the future, tool developers will be able to combine built-in tools with other community-contributed tools to support their high-level applications.

Amanda also includes a built-in **context mapping** tool that performs transformation with user-defined rules between namespaces. As previously explained, although we provide unified instrumentation points and interfaces across different backends in Amanda, the raw context remains backend-specific. State representations may have different formats in various frameworks, such as tensor layouts or operator naming conventions, as illustrated in Fig. 6. To achieve the backend framework portability, the raw context is transformed by intermediate layer mapping tools into a common format that can be consumed by the same instrumentation code. Thus, the transformation and high-level code can be implemented as portable tools, with the transformation tools being shared and reused by the high-level ones.

The built-in context mapping tools are used to support the development of those transformation tools by allowing users to only define transformation rules between namespaces instead of implementing individual tools. A namespace is a group of tags that specifies the framework name, version, and execution mode. For instance, an operator in the namespace "`tensorflow/1.13/graph`" indicates that it is executed on TensorFlow 1.13 in graph mode. Because the composition feature allows tools to update the operator's context and inject new state representations, this tool lets users register a mapping rule as a state representation transformation function. We illustrate an example of the Amanda mapping tool for the pruning tool in Sec. 4 with Lst. 6. With these mapping rules handling the transformation of raw context, the pruning tool can be applied to both PyTorch and TensorFlow backends. Notably, a mapping rule can register two functions that transform the computation state representations held by the context, one before and one after the analysis routines. As a result, users can implement cross-framework instrumentation tools rely on the high-level semantics of DNNs without dealing with the fragmented underlying details.

Minimizing instrumentation overhead via caching. In the Amanda-core, we introduce a set of cache techniques to optimize the instrumentation overhead in the eager mode and graph mode, respectively. The basic idea is to cache the redundant instrumentations and enable the reuse of repeated ones so that we can reduce irrelevant instrumentations. To elaborate, the instrumentation routines are represented as `action` objects defined in backend APIs, which will be introduced later in this paper. By logging the instrumentation actions of each operator, Amanda utilizes a cache mechanism to apply minimum DNN modifications and reduce redundant callbacks on operators. The effect of this action cache varies depending on the execution mode, as illustrated in Fig. 4. We will demonstrate the details later in Sec. 5.3.

Addressing the language disparity. Modern DNN backends are typically implemented in native languages such as C++ language, while present users with Python-level programming interfaces. The backend drivers employ runtime replacement of specific Python implementations for the purpose of instrumentation. To address this disparity, Amanda core incorporates a lightweight cross-language interceptor as a Python library, which dynamically replaces the specific implementation as needed. This interceptor effectively encapsulates Python functions or underlying native binding functions, eliminating the need for the driver to replace all references to such functions without necessitating a heap scan replacement. Furthermore, the design of the interceptor minimizes the introduction of cross-language trampolines, resulting in a low overhead for cross-language binding. For instance, even for small GPU kernels, the overhead is kept below 10 μs.

5.3 Amanda Backend

To reduce the case-by-case engineering efforts for different backends, we propose the implementation of a backend-dependent driver that offers simple and standardized callback mechanisms. As shown in Fig. 7, various backends may implement their callback mechanisms differently, but they all provide a common intermediate abstraction layer referred to as the backend interface. In other words, this interface definition isolates the Amanda drivers from the Amanda-core components, facilitating the reusability of the Amanda-core and enabling the extensibility of backend drivers. Amanda provides out-of-the-box driver support for TensorFlow's graph mode and PyTorch's eager mode. The drivers for other backends, such as MXNet [21] and MindSpore [20], can be implemented in the similar programming models, which are our future works.

Backend Interface. This thin abstraction layer defines the necessary functional or storage interfaces by drivers with `OpContext` and `Action` objects as shown in Fig. 7. The runtime computation states surrounding an operator are represented by an `OpContext` object, which encapsulates raw operator objects, contextual informations such as input tensors and output tensors as well as instrumentation metadata. This object is passed and rearranged through Amanda-cores and user-level APIs. Its member function, `trigger_callback()`, provides a raw callback interception interface, implemented by low-level drivers with an `i_point` argument that assigns the instrumentation point.

We unify the implementation of various instrumentation points, such as callbacks for forward and backward operators to a common interface with the `i_point` as the dispatching key. The `Action` object serves as the abstraction of an instrumentation action on the target DNN model. And all the instrumentation actions applied to an operator are tracked by its `OpContext`. The attributes of the `Action` object determine the type of instrumentation, the inserted callback function, and the required contextual tensors. Specifically, there are six types of instrumentation actions, which are the same as those in Lst. 3. Amanda will execute analysis routines and record all the actions generated by them. The registered actions are evaluated during the subsequent DNN executions as instrumentation routines.

Eager Mode Driver. In our implementation of the eager mode driver, we employ a technique called monkey-patching to wrap original operators with additional callbacks. The wrapped operator executes the instrumented actions before

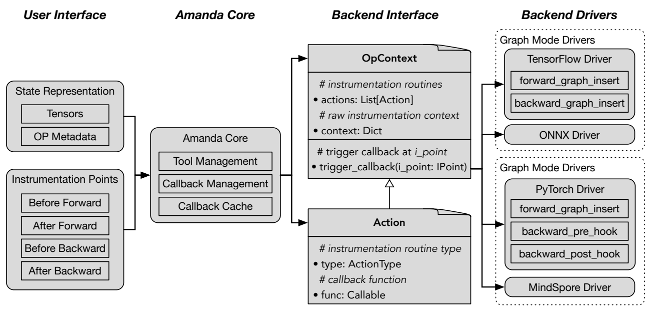

Fig. 7. Implementation of Amanda's unified user-level interface on different backends.

and after the original operator execution. However, there are two main challenges to be addressed.

The first challenge is the non-trivial task of patching all the operators, including the framework-native ones and user-defined ones, as there is no global reference for all the operators in the backend. To solve this problem, we snoop the operator registration process of the backend, ensuring that all registered operators, including dynamically linked ones defined by users, are correctly patched. However, this process also patches irrelevant operators, for which we use a cache mechanism to overcome, as described in Sec. 5.2. The second challenge is to correctly collect and map backward operators to their corresponding forward operators, which is essential for providing the necessary instrumentation context required by the backend APIs. In the eager mode, the backward operators are declared by the forward operators and are constructed as a linked list for subsequent execution. These backward operators do not have direct references to their corresponding forward operators. To overcome this challenge, we enhance the driver with the capability to incrementally track the backward operators. Specifically, after the execution of each forward operator, we traverse the backward computation graph and assign those newly declared backward operators to the current forward operator. In particular, we implement this tracking process using the raw callback mechanism as a post-execution hook.

We then demonstrate the cache mechanism in the eager mode drivers. By logging the instrumentation routines as action on each operator, the action cache eliminates unnecessary callbacks for patched operators without cached actions, resulting in significant performance improvements. To ensure the instrumentation points are provided for all dynamic operators during the initial execution, Amanda patches all operators, including irrelevant ones. By caching the instrumentation routines as action objects, operators that are not instrumented by users are switched back to their vanilla counterparts, and heavy analysis routines are reused. This

Tasks	Projects	Type	Graph Mode		Eager Mode		Amanda Tool	
			Interface	Portable	Interface	Portable	Interface	Portable
Graph Tracing	Built-in	Analysis	Graph	All	Module Hook	Refactor	Instrumentation	All
FLOPs Profiling	[6, 8, 10, 15]	Analysis	Graph	All	Module Hook	Refactor	Instrumentation	All
Effective Path	[32, 70]	Analysis	Graph, Source Modification	No	Module Hook	No	Instrumentation	All
Weight Pruning	[40, 48, 82]	Optimization	Session Hook	No	Module Parameter	Refactor	Instrumentation	All
Quantization Training	[18, 30, 56]	Optimization	Source Modification	No	Module Hook	Refactor	Instrumentation	All

Table 3. Representative DNN analysis and optimization tasks that can be implemented as Amanda tools using its instrumentation interface comparison with their native implementations in different execution modes.

strategy eliminates redundant callbacks for empty instrumentation, thereby reducing the instrumentation overhead.

Graph Mode Driver. We implement this driver by modifying the computation graph with operator insertions. The driver retrieves the computation graph from the backend runtime and replaces it with the modified version. The analysis routine is invoked during this rewriting process, while the instrumentation routines are added to the graph as operators and executed at runtime. One major challenge of this rewriting approach is to ensure the isolation between the instrumentation code and the original graph operators. Specifically, the instrumented code should not be visible to the operators of the original graph. For example, if the instrumentation code adds a parameter to the graph to maintain an internal state, a graph saver operator from the original computation graph may inadvertently include this parameter in the checkpoint, which is an undesired behavior.

We develop a graph switching mechanism to effectively address the above issue. The driver maintains two versions of the graph: the original instance and the instrumented instance. The instrumented graph is utilized for backend runtime execution, while the original “vanilla" graph is used by operators to access the graph. To seamlessly switch between the two graphs during runtime, we wrap the execution call with a mechanism that updates the instrumented graph with the latest computation states from the raw graph and applies instrumentation actions. Additionally, after the instrumented execution, the raw graph's computation states are updated by the instrumented graph.

On the other hand, naively implementing the above graph modification could be rather expensive in graph execution backends. As such, we adopt the cache mechanism at a coarse-grained graph level, where all actions applied to the computation graph are cached as an instrumented graph. The graph object is hashed and reused across executions until further modifications are made. The heavy and time-consuming graph rewrite/switch operation only occurs when new instrumentation routines update the graph.

6 Evaluation: Amanda Tools

In this section, we evaluate the Amanda framework by developing multiple representative several use cases. Through this comprehensive evaluation, we aim to demonstrate the key advantages of Amanda, including its generality to support mainstream DNN analysis and optimization tasks, the capability of integrating additional low-level instrumentation framework, user-friendly and consistent interfaces across different execution backends, and low execution overhead.

6.1 Generality: Supporting Various Use Cases

To show the feasibility of Amanda to support the analysis and optimization of DNN models, we evaluate Amanda with several common and representative use cases including general computation graph tracing, FLOPs profiling, DNN interpretability tool effective path, model pruning and model quantization. Tbl. 3 summarizes these tasks.

The first general computation graph tracing task traces the execution operators and computation graph of a DNN model. It is often used to debug a model during development or get the graph structure for further processing. As introduced in Sec. 5.2, we make graph tracing a built-in tool to be easily used by other Amanda tools. Profiling is to trace and count the runtime states of DNN operators. It is important for further DNN optimization of all levels, such as compiler-level optimization [22], system-level scheduling [89], or even hardware-level architecture design [76]. Here, we show a common FLOP profiler that evaluates the computational complexity of a DNN model. This is a very common demand and a lot of projects implement it as part of its utility function [33]. Though we can get such performance counters with GPU-provided APIs, they are difficult to use considering the challenges we stressed in Sec. 2. The GPU performance profiling tool makes such a process easy and portable.

There is also a line of works that study the interpretability of DNN models. For example, prior work [32, 70] extracts a sub-structure of the DNN model under certain input called the effective path to understand the model's behavior. Extracting the effective path from various models requires both the forward and backward computation graph, and takes a significant amount of manual refactoring and even source code modification. For optimization tasks, we use weight pruning and quantization-aware training discussed in Sec. 2.

As shown in Tbl. 3, we implement these common use cases to show the generality of Amanda supporting popular tasks.

Project	Type	User Tool					Amanda tool	
		Backend	Interface	Supported Networks	LoC	Acc	LoC	Acc
Tile Wise Pruning[40]	Static	Tensorflow	Session Hook	VGGs, BERT	1203	76.7	213	76.7
Dynamic Channel Pruning[33]	Dynamic	PyTorch	Source Modification	VGG19, ResNet34, SquezzeNet	387	70.7	115	70.7
Activation Pruning[78]	Dynamic	PyTorch	Source Modification	ResNets	650	77.1	193	76.5
Attention Pruning[39]	Dynamic	PyTorch	Source Modification	BERT, Roberta, DistillBERT, ALBERT	1105	83.2	179	83.2
APEX Vector Wise Pruning[7, 85]	Static	PyTorch	Module Hook	Models with Module API	499	76.5	279	76.2

Table 4. Evaluation of pruning projects from the community and the re-produced Amanda tool in line of codes (LoC). Accuracy (Acc) is evaluated with the ResNet50 model [51] on the ImageNet dataset [25] and BERT-base model [26] on SQuAD-V2 question-answering dataset [71].

For the tasks for which we cannot find implementations in both execution modes, we provide a prototype implementation as a baseline. We can find that implementing these use cases requires ad-hoc interfaces with reference to the instrumentation points and state representations they require. Meanwhile, these developed tools are often not portable and tied to several specific DNN models, which require a significant amount of engineering efforts or direct source modification to adapt them to different DNN models. With Amanda framework, users can develop such tools without considering the details of a specific DNN model, and construct an instrumentation tool that is portable to all models. Tbl. 3 shows that while different projects prefer various backends/interfaces, Amanda framework's instrumentation abstraction reduces the developers' learning bar to deal with the complexity of backends and interfaces but concentrates on the task itself. We give a detailed demonstration of two Amanda tools in the following.

6.2 Developer Friendly: Network Pruning

We then demonstrate how Amanda benefits developers on actual projects. As we have demonstrated in Sec. 2, developing network pruning use cases faces a series of problems caused by current fragmented DNN execution backends.

Tbl. 4 shows several representative pruning projects that we collect from the DNN algorithm research community including different pruning patterns, pruning strategies, and object tensors. With fragmented interfaces and network source code, developers often write ad-hoc programs for these similar pruning algorithms. We find that it is even common to directly modify the source code of every supported network to adopt their proposed pruning method. We can find that developers use different implementations according to the algorithms, target DNN model, and even their preference. This is extremely user-unfriendly and non-extendable. In contrast, Amanda can support all these pruning tasks with a unified instrumentation abstraction, which is to simply insert the pruning operators with Amanda APIs and apply the tool during DNN training. As such, the developer can handle the unified state representation and apply the pruning tool to all models without extra effort.

Tbl. 4 shows that developing the same project with Amanda framework requires much fewer lines of codes (LoC). This is because current user implementations spend many codes dealing with the DNN model definition and execution codes, which scales directly as the number of DNN models increases. Meanwhile, there is a large fraction of codes used to deal with the instrumentation process, such as fetching required tensors or applying the pruning function to the training loop.

As the Amanda instrumentation tool is decoupled from the original DNN definition and execution codes, the lines of codes in different use cases are greatly reduced when applied to multiple DNN models. In addition, Amanda does not change the execution semantics of a specific pruning task. We validate this behavior by showing that Amanda-based implementation achieves the same level of model accuracy as the original implementation. APEX [7] is a general pruning tool supporting the vector wise pruning algorithm provided by TensorCore [24] hardware. So Amanda does not have as much LoC reduction as it does on other projects. It is portable to different DNN models but only supports a fixed sparsity pattern. Besides, APEX only supports networks defined with PyTorch's module API. Users may have to refactor a target DNN model with this module API in order to use this tool.

6.3 Synergy with Low-level Kernel Instrumentation

As introduced in Sec. 3, Amanda is designed for operator-level or coarser-grained subgraph (e.g. module in PyTorch) instrumentation points. For fine-grained instrumentation such as kernel- and binary-level, the design of Amanda is

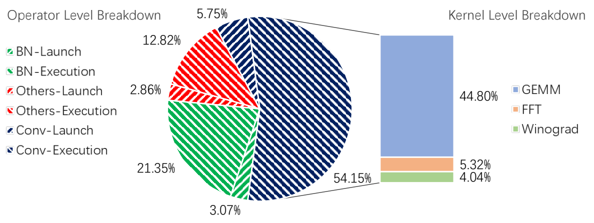

Fig. 8. Operator level GPU execution time breakdown with kernel-level profiling.

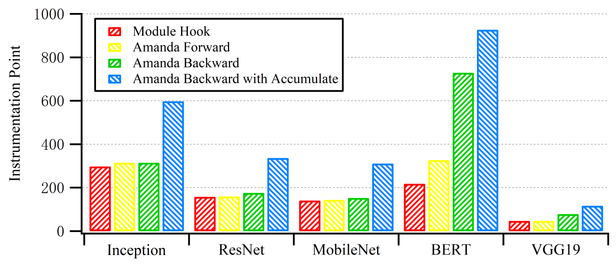

Fig. 9. Evaluation results of eager mode op coverage of PyTorch module hook and Amanda.

naturally compatible to existing mature and vendor-specific solutions such as Pin [61], NVBit [79], and CUPTI [5].

To demonstrate the synergy instrumentation relationship between Amanda and CUPTI, we have developed an Amanda tool that supports the profiling of underlying GPU kernels. Profiling the hardware performance counters of each DNN operator is essential for many optimizations [40, 65, 89]. Some DNN execution engines [10, 12] provide integrated GPU profiling support to collect low-level information such as kernel execution time and memory utilization. However, these integrated profilers only provide overall statistics and are not customizable for low-level GPU APIs. As a result, it requires serious human effort to refactor the DNN source codes to get the operator isolated and profiled.

With the use of Amanda, such CUPTI API can be instrumented into the DNN model easily. We can simply declare the hardware tracer and counter before operator execution and perform clean-up after operator execution to get operator-level performance counter metrics. The available metrics include kernel launch time, kernel execution time, memory access, and so on. Fig. 8 shows the overall GPU time breakdown of a ResNet50 model. With the operator level instrumentation points of Amanda, we can even aggregate the low-level kernel results at the operator level We use the diminant convolution operator as an example. While `im2col`-based [88] kernels are used for most cases, FFT and Winograd algorithms are also used for several cases. That information is useful for operator or compiler developers to improve execution efficiency. Amanda makes the profiling of low-level metrics fully customizable and easy-to-do.

6.4 Instrumentation Point Coverage

We then evaluate the instrumentation point coverage of Amanda compared to the module hook interface that is commonly used in PyTorch [68]. Module hooks are provided in PyTorch to instrument its execution to inspect runtime and backward phase information. However, as demonstrated in the aforementioned sections, it suffers from the module declaration limitation. As determined by the DNN model source declaration, many ops are not reachable under various circumstances. It is a common effort for developers to conduct source code modification or refactor the DNN model.

Fig. 9 shows the number of instrumentation points covered by PyTorch module hook and Amanda. In all cases except VGG19 [74], there are ops missed by the module hook in the forward phase. This is because there are multiple operators in one module object. For complicated DNN model BERT [26], the module hook fails to capture over 100 of the total 327 forward operators. It also misses more operators in the backward phase because a forward operator may call multiple backward operators for gradient calculation. In that case, the module hook only captures the entrance and exit instrumentation point for those backward operators. As a result, it fails to capture over four hundred backward operators as shown in Fig. 9.

This omission can lead to incorrect results for analysis tools. Take the add operator of skip connections in ResNet [51] as an example. They are declared by the functional APIs, and hence omitted by module hooks, which leads to wrong results in effective path extraction task [70]. Another thing worth mentioning is the gradient accumulation ops in the backward pass, all of which are omitted by module hooks. Instead, Amanda provides instrumentation points for them, which leverages handy manipulations on gradient accumulation operation.

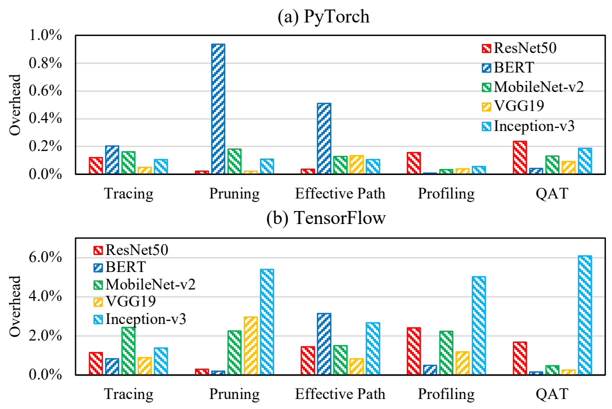

Fig. 10. Overhead of Amanda on the use cases introduced in Sec. 6 with several common neural network models.

6.5 Instrumentation Overhead

To make the instrumentation framework applicable, its overhead should be kept at a low and acceptable level. We evaluate Amanda's overhead on these use cases and further show the effects of the cache management mechanism.

Fig. 10 shows the overhead of various models and use cases. Overall, the overhead of Amanda is less than 1% in eager mode and 7% in graph mode. This low overhead guarantees the instrumentation framework is applicable in real scenarios. In eager mode, pruning and effective path tools on

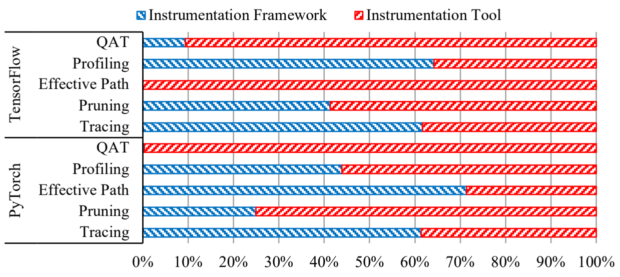

Fig. 11. The normalized execution time breakdown for each use case by Amanda framework and tools.

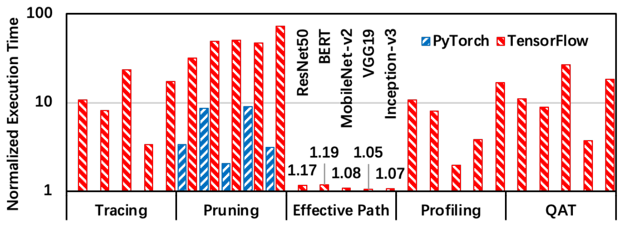

Fig. 12. Normalized overall execution latency of Amanda instrumentation tools without cache mechanism normalized to its optimized counterpart. Larger normalized latency represents better optimization by caching.

the BERT model have a larger overhead. This is because they contain forward and backward subgraph matching which is slow. In graph mode, the inception-v3 model [77] has the highest overhead for most cases. This is because this model implementation consists of many operators.

Fig. 11 shows the breakdown of the overhead of Amanda framework and use-case itself (Amanda tool). There is a large range of varieties among these five different Amanda tools. For example, the QAT (quantization-aware training) tool contains a computationally heavy instrumentation routine that performs extra calculations with runtime tensors. As such, the instrumentation framework overhead is negligible compared to the tool itself. We observe a similar trend for the effective path tool [70]. However, this tool requires the graph structure and invokes the built-in graph tracing in PyTorch, and we account this built-in graph tracing tool's overhead in Amanda framework.

We then evaluate the effectiveness of our cache mechanism in Fig. 12. We normalize the overall execution time of instrumentation tools without cache mechanism to its optimized version. In the eager mode, the analysis routines are cached after their first execution. The static pruning use case is the only one that contains a heavy analysis routine and benefits the most (over 2× larger latency). In the graph mode, the graph rewriting procedure is cached. As such, most use cases benefit from the cache mechanism. Similar to the eager mode, the pruning use case has the most speedup ratio when enabling the cache mechanism as the analysis routine could also be optimized in graph mode. Compared with the baseline without enabling the cache mechanism, Amanda achieves 72.6× speedup at most and 17.1× speedup on average, showing the effectiveness of our cache mechanism.

We further evaluate the GPU memory usage of Amanda framework across various input sizes to further analyze its associated overhead. To achieve this, we adopt a tracing use case that spans all operators in the DNN processing pipeline. We present the results with ResNet50 with input size 224 × 224 and Transformer with input sequence length of 128, serving as a representative setting. Notably, TensorFlow incorporates an intrinsic mechanism for device memory management, which declares maximum memory for internal scheduling. In our experimentation, we deactivate this feature, opting to execute the instrumentation tools with its dynamic memory allocation method. Despite this, it remains pertinent to highlight that Amanda retains seamless compatibility with TensorFlow's foundational memory management mechanism. This is due to the fact that Amanda integrates a fully modified computation graph into the runtime without disrupting the existing memory management protocols. The memory footprint breakdown of the instrumentation tools is shown with Fig. 13. We can find that Amanda generally has a minor memory overhead which is less than 5% for most cases. For input tensors featuring a larger batch size, Amanda framework has a reduction in memory footprint overhead because of its fixed management overhead. However, when evaluating the Transformer model within the TensorFlow backend, it is evident that overhead often approximates or surpasses 10% for smaller batch sizes of 1 or 2. This phenomenon is driven by the model's relatively small size and input dimensions, culminating in a relatively substantial instrumentation management overhead.

7 Related Work and Discussion

In this section, we discuss other techniques and interfaces for DNN application development. We further analyze other independent DNN tools or compilers and their collaboration with the instrumentation tools of Amanda.

Eager Mode Hooks. In PyTorch's eager mode, developers can utilize runtime hooks to alter the runtime behavior or capture runtime state representations. For instance, backward module hooks can be used to retrieve the activation tensors and their gradient in both forward and backward passes. Although these hooks support the instrumentation of any module, they cannot be applied to operators that are not wrapped as modules, which limits their universality. Furthermore, functional operator APIs [11] cannot be covered by PyTorch's module hooks, restricting their applicability when the model employs these APIs.

Graph Transformation. Graph transformation is a technique used to transform the computational graph in graph mode. For instance, TensorFlow's session hooks can be used to achieve tracing by providing a session hook when an estimator submits a graph to the session, enabling extra fetches to be attached for instrumentation. However, legal graph transformation is limited since TensorFlow's graph is append-only for users. Without the help of internal graph APIs, tracing cannot be implemented by the users. Additionally, TensorFlow's graph will seal after submission, which breaks the tracing tool when there are multiple session submissions, such as when initializing parameters before inference. Grappler [4], which is the default graph optimization system of TensorFlow, can be an alternative for instrumentation, despite not being designed for this task. However, Grappler only provides C++ APIs, raising the barrier to use, and it is not compatible with TensorFlow Eager. FX [2], a PyTorch toolkit for module transformation, uses symbolic tracing to capture the graph representation inside the module and provides APIs to arbitrarily instrument the module. Nevertheless, it is not fully compatible with eager mode since some dynamic graphs cannot be symbolically traced, and the instrumentation API is verbose since users are manipulating the graph AST directly.

High-level Tools. There are also many high-level DNN tools, such as Torch-Pruning [31] for pruning, NNI [64] for model compression, TensorBoard [3] for visualization, and many others. These tools are primarily higher-level analysis or optimization tools, whereas Amanda serves as a lower-level instrumentation infrastructure. As shown in Sec. 6.1, the semantics of these tools can be easily implemented with Amanda's fundamental instrumentation abstraction. Specifically, TensorBoard can also instrument the operators and profile metrics of various DNN models. However, TensorBoard is limited to profiling and lacks customization options for non-trivial use cases such as pruning or quantization. Furthermore, we introduced how Amanda interacts with lower-level instrumentation tools to deliver cross-layer instrumentation features in Sec. 6.3. Amanda can also cooperate with the TensorBoard visualization tool to reuse its well-defined functionalities. For example, the tracing tool developed with Amanda dumps the trace file in JSON format, which can also be visualized with TensorBoard.

Synergy with DNN Compilers. There have emerged many DNN compilers [59, 87], such as TVM [22], TorchDynamo [16], optimizing the execution of DNN models. These compilers may change the execution flow, fuse adjacent operators, and dispatch operators for efficient kernel implementation. With these model-level optimizations, certain instrumentation points could be removed since the low-level compiler can modify the execution graph or operators. One potential solution is to employ an intermediate level that maintains the relationship between the remaining instrumentation points and the original ones. Alternatively, one could also disable the compiler optimizations, similar to how a program debugger disables certain optimizations in favor of debugging rather than performance.

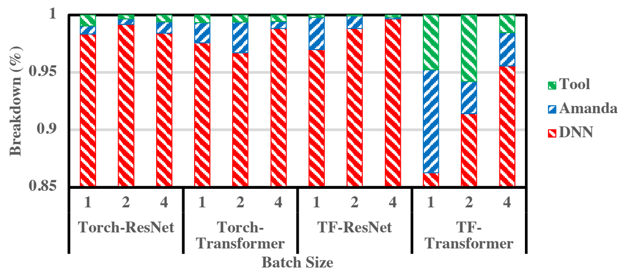

Fig. 13. Memory footprint overhead breakdown of tracing tool with Amanda of different input batch sizes

8 Conclusion

In this study, we propose to utilize the instrumentation programming model to support the emerging DNN analysis and optimization tasks, for which developers and researchers currently adopt ad-hoc and fragmented techniques. Throughout the development, we explored several abstraction models, including maintaining an internal operator set or conducting namespace-to-namespace operator conversion. Ultimately, we discovered that an operator abstraction with context mapping is the optimal approach for ensuring the extensibility of the framework while providing a unified interface. As such, we build Amanda which allows users produce portable, reusable and composable tools with a unified instrumentation interface. Our framework has a low execution overhead and is designed with a clear layered architecture, making it easy to adapt to new backends. We believe that these features will make our Amanda framework a practical and open-source infrastructure for the community. Ultimately, we look forward to providing long-term support of the framework, including bringing it to new backends and further optimization of its efficiency.

Acknowledgment

This work was supported by the National Key R&D Program of China under Grant 2021ZD0110104, the National Natural Science Foundation of China (NSFC) grant (62222210, U21B2017, 61832006, and 62072297). We would like to thank the anonymous reviewers for their constructive feedback for improving the work. We also thank our shepherd, Xipeng Shen, for his ongoing support and guidance during the revision process. We also thank Changming Yu, Renyang Guan, Zihan Liu, Cong Guo, and many other colleagues for testing Amanda framework and providing useful feedback during its design. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] 2021. ONNX Runtime. https://onnxruntime.ai/.

[2] 2021. PyTorch FX. https://pytorch.org/docs/stable/fx.html.

[3] 2021. TensorBoard. https://tensorflow.org/tensorboard.html.

[4] 2021. TensorFlow Grappler. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/core/grappler.

[5] 2021. CUDA Profiling Tools Interface. https://docs.nvidia.com/cupti/index.html.

[6] 2021. FLOPs Counter. https://github.com/sovrasov/flops-counter.pytorch.

[7] 2021. NVIDIA APEX. https://github.com/NVIDIA/apex.

[8] 2021. OneFlow OpCounter. https://github.com/Oneflow-Inc/flow-OpCounter.

[9] 2021. TensorFlow Model-Optimization Sparsity. https://github.com/tensorflow/model-optimization/tree/v0.7.3/tensorflow_model_optimization/python/core/api/sparsity.

[10] 2021. TensorFlow Profiler. https://github.com/tensorflow/profiler.

[11] 2021. Torch Functional. https://pytorch.org/docs/stable/nn.functional.html.

[12] 2021. Torch Profiler. https://pytorch.org/docs/stable/profiler.html.

[13] 2021. Torch Prune. https://github.com/pytorch/pytorch/blob/master/torch/nn/utils/prune.py.

[14] 2021. Torch Quantization. https://pytorch.org/docs/stable/quantization.html.

[15] 2021. TorchProfile. https://github.com/zhijian-liu/torchprofile.

[16] 2021. TorchDynamo. https://pytorch.org/docs/stable/dynamo/index.html.

[17] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[18] Yash Bhalgat, Jinwon Lee, Markus Nagel, Tijmen Blankevoort, and Nojun Kwak. 2020. LSQ+: Improving low-bit quantization through learnable offsets and better initialization. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2020, Seattle, WA, USA, June 14-19, 2020*. Computer Vision Foundation / IEEE, 2978–2985. https://doi.org/10.1109/CVPRW50498.2020.00356

[19] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What is the state of neural network pruning? *Proceedings of machine learning and systems* 2 (2020), 129–146.

[20] Lei Chen. 2021. *Deep Learning and Practice with MindSpore*. Springer Nature.

[21] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[22] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[23] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

[24] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 tensor core GPU: Performance and innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[27] Xuanyi Dong and Yi Yang. 2019. Network Pruning via Transformable Architecture Search. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 759–770. http://papers.nips.cc/paper/8364-network-pruning-via-transformable-architecture-search

[28] Yinpeng Dong, Hang Su, Jun Zhu, and Bo Zhang. 2017. Improving Interpretability of Deep Neural Networks With Semantic Information. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[29] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=YicbFdNTTy

[30] Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. 2020. Learned Step Size quantization. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=rkgO66VKDS

[31] Gongfan Fang, Xinyin Ma, Mingli Song, Michael Bi Mi, and Xinchao Wang. 2023. Depgraph: Towards any structural pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 16091–16101.

[32] Yiming Gan, Yuxian Qiu, Jingwen Leng, Minyi Guo, and Yuhao Zhu. 2020. Ptolemy: Architecture support for robust deep learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 241–255.

[33] Xitong Gao, Yiren Zhao, Łukasz Dudziak, Robert Mullins, and Chengzhong Xu. 2018. Dynamic channel pruning: Feature boosting and suppression. *arXiv preprint arXiv:1810.05331* (2018).

[34] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. 2019. Estimation of energy consumption in machine learning. *J. Parallel and Distrib. Comput.* 134 (2019), 75–88.

[35] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2021. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630* (2021).

[36] Andreas Griewank et al. 1989. On automatic differentiation. *Mathematical Programming: recent developments and applications* 6, 6 (1989), 83–107.

[37] Yue Guan, Jingwen Leng, Chao Li, Quan Chen, and Minyi Guo. 2020. How Far Does BERT Look At: Distance-based Clustering and Analysis of BERT ′ s Attention. *arXiv preprint arXiv:2011.00943* (2020).

[38] Yue Guan, Zhengyi Li, Jingwen Leng, Zhouhan Lin, and Minyi Guo. 2022. Transkimmer: Transformer Learns to Layer-wise Skim. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7275–7286.

[39] Yue Guan, Zhengyi Li, Zhouhan Lin, Yuhao Zhu, Jingwen Leng, and Minyi Guo. 2022. Block-skim: Efficient question answering for transformer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 10710–10719.

[40] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse dnn models without hardware-support via tile-wise sparsity. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[41] Cong Guo, Yuxian Qiu, Jingwen Leng, Xiaotian Gao, Chen Zhang, Yunxin Liu, Fan Yang, Yuhao Zhu, and Minyi Guo. 2022. SQuant: On-the-fly data-free quantization via diagonal hessian approximation. *arXiv preprint arXiv:2202.07471* (2022).

[42] Cong Guo, Yuxian Qiu, Jingwen Leng, Chen Zhang, Ying Cao, Quanlu Zhang, Yunxin Liu, Fan Yang, and Minyi Guo. 2022. Nesting Forward Automatic Differentiation for Memory-Efficient Deep Neural Network Training. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 738–745.

[43] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2023. OliVe: Accelerating Large Language Models via Hardware-friendly Outlier-Victim Pair Quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[44] Cong Guo, Chen Zhang, Jingwen Leng, Zihan Liu, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2022. Ant: Exploiting adaptive numerical data type for low-bit deep neural network quantization. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1414–1433.

[45] Yiwen Guo, Anbang Yao, and Yurong Chen. 2016. Dynamic Network Surgery for Efficient DNNs. *CoRR* abs/1608.04493 (2016). arXiv:1608.04493 http://arxiv.org/abs/1608.04493

[46] Sean Hallgren. 1994. *Linear congruential generators over elliptic curves*. Citeseer.

[47] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[48] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[49] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. 2021. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[52] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*. 1389–1397.

[53] Sara Hooker, Dumitru Erhan, Pieter-Jan Kindermans, and Been Kim. 2019. A Benchmark for Interpretability Methods in Deep Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 9734–9745. https://proceedings.neurips.cc/paper/2019/hash/fe4b8556000d0f0cae99daa5c5c5a410-Abstract.html

[54] Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146* (2018).

[55] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *CoRR* abs/1712.05877 (2017). arXiv:1712.05877 http://arxiv.org/abs/1712.05877

[56] Sambhav R. Jain, Albert Gural, Michael Wu, and Chris Dick. 2020. Trained Quantization Thresholds for Accurate and Efficient Fixed-Point Inference of Deep Neural Networks. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/295.pdf

[57] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=Vfs_2RnOD0H

[58] Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems*. 598–605.

[59] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 388–401.

[60] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2018. Rethinking the Value of Network Pruning. *CoRR* abs/1810.05270 (2018). arXiv:1810.05270 http://arxiv.org/abs/1810.05270

[61] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[62] Divyam Madaan and Sung Ju Hwang. 2019. Adversarial Neural Pruning. *CoRR* abs/1908.04355 (2019). arXiv:1908.04355 http://arxiv.org/abs/1908.04355

[63] Paul Michel, Omer Levy, and Graham Neubig. 2019. Are Sixteen Heads Really Better than One? *arXiv preprint arXiv:1905.10650* (2019).

[64] Microsoft. 2021. *Neural Network Intelligence*. https://github.com/microsoft/nni

[65] Saiful A Mojumder, Marcia S Louis, Yifan Sun, Amir Kavyan Ziabari, José L Abellán, John Kim, David Kaeli, and Ajay Joshi. 2018. Profiling dnn workloads on a volta-based dgx-1 system. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 122–133.

[66] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. 2018. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 580–595.

[67] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[68] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[69] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 891–905.

[70] Yuxian Qiu, Jingwen Leng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2019. Adversarial Defense Through Network Profiling Based Path Extraction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4777–4786.

[71] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[72] Alex Renda, Jonathan Frankle, and Michael Carbin. 2020. Comparing Rewinding and Fine-tuning in Neural Network Pruning. *arXiv preprint arXiv:2003.02389* (2020).

[73] Justyna Sarzynska-Wawer, Aleksander Wawer, Aleksandra Pawlak, Julia Szymanowska, Izabela Stefaniak, Michal Jarkiewicz, and Lukasz Okruszek. 2021. Detecting formal thought disorder by deep contextualized word representations. *Psychiatry Research* 304 (2021), 114135.

[74] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.1556

[75] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. 2019. Low-memory neural network training: A technical report. *arXiv preprint arXiv:1904.10631* (2019).

[76] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[77] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[78] Thomas Verelst and Tinne Tuytelaars. 2020. Dynamic convolutions: Exploiting spatial sparsity for faster inference. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2320–2329.

[79] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 372–383.

[80] Yulin Wang, Rui Huang, Shiji Song, Zeyi Huang, and Gao Huang. 2021. Not all images are worth 16x16 words: Dynamic transformers for efficient image recognition. *Advances in Neural Information Processing Systems* 34 (2021), 11960–11973.

[81] Yulong Wang, Xiaolu Zhang, Lingxi Xie, Jun Zhou, Hang Su, Bo Zhang, and Xiaolin Hu. 2019. Pruning from Scratch. *CoRR* abs/1909.12579 (2019). arXiv:1909.12579 http://arxiv.org/abs/1909.12579

[82] Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. 2019. Balanced sparsity for efficient dnn inference on gpu. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 5676–5683.

[83] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*. IEEE, 36–39.

[84] Zixing Zhang, Jürgen Geiger, Jouni Pohjalainen, Amr El-Desoky Mousa, Wenyu Jin, and Björn Schuller. 2018. Deep learning for environmentally robust speech recognition: An overview of recent developments. *ACM Transactions on Intelligent Systems and Technology (TIST)* 9, 5 (2018), 1–28.

[85] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. 2021. Learning n: m fine-grained structured sparse neural networks from scratch. *arXiv preprint arXiv:2102.04010* (2021).

[86] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. 2016. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *CoRR* abs/1606.06160 (2016). arXiv:1606.06160 http://arxiv.org/abs/1606.06160

[87] Yangjie Zhou, Jingwen Leng, Yaoxu Song, Shuwen Lu, Mian Wang, Chao Li, Minyi Guo, Wenting Shen, Yong Li, Wei Lin, et al. 2023. uGrapher: High-Performance Graph Operator Computation via Unified Abstraction for Graph Neural Networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 878–891.

[88] Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 214–225.

[89] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately Estimating the Efficacy of Optimizations for {DNN} Training. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 337–352.