Relay: A New IR for Machine Learning Frameworks

Jared Roesch jroesch@cs.uw.edu
Steven Lyubomirsky sslyu@cs.uw.edu
Logan Weber weberlo@cs.uw.edu
Josh Pollock joshpoll@cs.uw.edu
Marisa Kirisame jerry96@cs.uw.edu
Tianqi Chen tqchen@cs.uw.edu
Zachary Tatlock ztatlock@cs.uw.edu

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle, WA, USA

Abstract

Machine learning powers diverse services in industry including search, translation, recommendation systems, and security. The scale and importance of these models require that they be efficient, expressive, and portable across an array of heterogeneous hardware devices. These constraints are often at odds; in order to better accommodate them we propose a new high-level intermediate representation (IR) called Relay. Relay is being designed as a purely-functional, statically-typed language with the goal of balancing efficient compilation, expressiveness, and portability. We discuss the goals of Relay and highlight its important design constraints. Our prototype is part of the open source NNVM compiler framework, which powers Amazon's deep learning framework MxNet.

CCS Concepts • **Computer systems organization** → **Architectures**; *Neural networks*; *Heterogeneous (hybrid) systems*; • **Software and its engineering** → **Compilers**; **Domain specific languages**; • **Computing methodologies** → *Machine learning*;

Keywords intermediate representation, machine learning, compilers, differentiable programming

ACM Reference Format:
Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A New IR for Machine Learning Frameworks. In *Proceedings of 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3211346.3211348

MAPL'18, June 18, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5834-7/18/06…$15.00
https://doi.org/10.1145/3211346.3211348

1 Introduction

Machine learning (ML) has dramatically reshaped computer vision [17, 21], natural language processing, robotics [14], and computational biology and is continuing to gain traction in new areas, including program synthesis [6]. Deep learning (DL) in particular has driven progress in these areas and is now powering diverse services in industry, including search, translation, recommendation systems, and security. Hardware diversity is growing almost as quickly. DL models are deployed not only in the cloud, but also on a myriad of devices, from off-the-shelf CPUs and GPUs to specialized smartphone chips and IOT edge devices.

The rise of GPUs for DL compute has made deep learning both possible and scalable for many tasks, but a new generation of applications with greater compute demands is already upon us. In an attempt to keep up with the ML community's insatiable desire for fast and efficient computation, researchers and industry professionals have introduced a new generation of diverse hardware accelerators and specialized architectures such as FPGAs and Google's TPU [19].

In order to properly schedule, train, and deploy models in a massively distributed, parallel, and heterogeneous computing environment, scalable systems must be built on top of specialized hardware to address the challenges of ever-growing available datasets. Yet even with today's tools, modern systems struggle to satisfy machine learning's increasing computational demands. Users must balance interdependent trade-offs at all levels of the DL hardware/software stack. New hardware might mean dramatically redesigning model architectures, tweaking precision, tuning hyperparameters, rewriting kernels, and FPGA or ASIC designs [9]. Adapting applications and systems to early accelerators demands substantial rethinking, redesign, and re-implementation to achieve the best performance. Indeed, multiple projects have already been undertaken to address this problem, such as Nvidia's Axon, Tensor Comprehensions [35], and Halide [30]. As the tools for building state-of-the-art ML systems grow more complex and varied, it is imperative that models, hardware, and systems be co-designed and tuned by applying ideas from synthesis and learning.

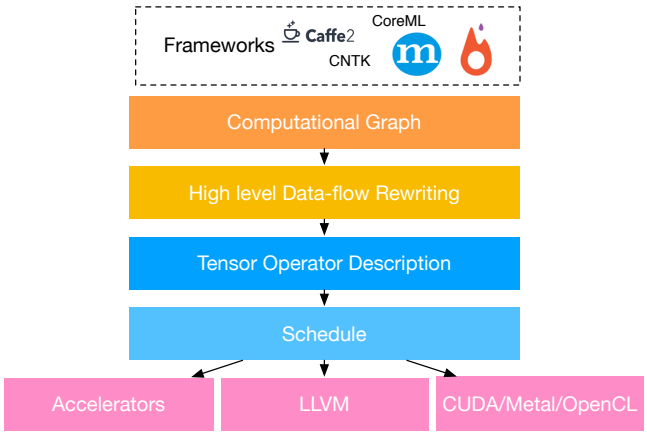

Figure 1. The present TVM stack, as presented in [9].

1.1 Existing High-Level Representations

An end-to-end stack and programming model for heterogeneous hardware would help satisfy the increasing demands for compute resources and enable machine learning with specialized accelerators. Research on the lower levels of such a stack has been realized in the form of the Tensor Virtual Machine (TVM), a hierarchical multi-tier compiler stack and runtime system for deep learning, depicted in Figure 1.

This work is focused on redesigning the top level of the TVM stack, as depicted in Figure 2. This level, known as NNVM, is based on computation graphs, the most popular representation of differentiable computation.

Machine learning often relies on computations that are differentiable, i.e., computations where it is possible to compute a mathematical derivative. In order to guarantee this property for users' programs, existing frameworks have limited programs' computational expressivity. Frameworks like TensorFlow represent differentiable computation using static graphs, which are dataflow graphs with a fixed topology.

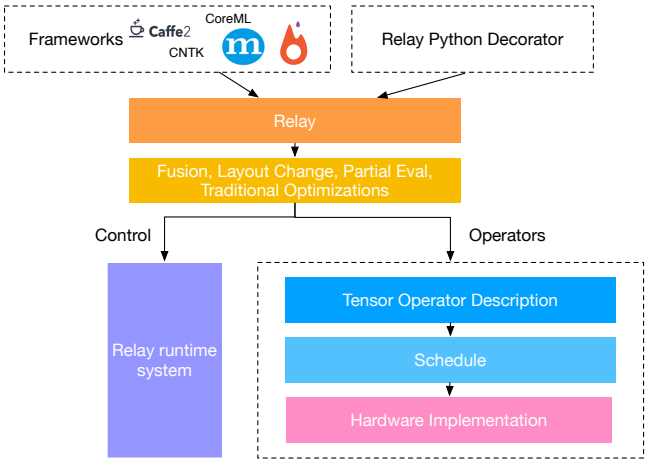

Figure 2. The new TVM stack integrated with Relay.

These graphs are easy to optimize but require users to construct programs in a deeply-embedded domain-specific language (eDSL) without high-level abstractions like functions.

A more expressive style popularized by imperative frameworks like Chainer, PyTorch, and Gluon allows the construction of graphs with dynamic topologies that can depend on runtime data and support differentiation of most imperative computations. This expressivity is convenient for the user but has limited the ability for existing frameworks to optimize user-defined graphs. Moreover, PyTorch's model requires a Python interpreter, making deployment to new accelerators and FPGAs extremely challenging.

In summary, static graphs are easy to optimize but lack the expressivity found in higher-level languages; dynamic graphs provide this missing expressivity but introduce new compilation and execution challenges, especially on heterogeneous hardware and FPGAs.

1.2 Relay

This work proposes Relay, a new high-level intermediate representation (IR) and language designed to balance efficient compilation, expressiveness, and portability by combining insights from the approaches of static graphs and dynamic graphs under the aegis of a functional programming language.

That is, we design Relay not from the perspective of a computation graph but from that of a programming language for differentiable computation. This PL perspective will allow Relay to exploit decades of research in functional programming, type systems, synthesis, rewrite systems, and classical compiler techniques. Our intent in future work is to demonstrate that these techniques and features will reduce the cost of targeting new accelerators and enable more optimizations to improve training and inference time, energy consumption, and space utilization.

This paper presents work in progress towards:

- Relay, a new differentiable language for expressing machine learning models.
- Higher-order automatic differentiation of Relay programs.
- A shape-dependent tensor type system for Relay.
- A baseline evaluator, and type-specialized operator compiler built on TVM.

2 Background and Related Work

Current DL IRs, including NNVM's current representation, are heavily inspired by dataflow programming and related computation graph abstractions made popular by previous frameworks.

For example, TensorFlow [4] is an iteration on previous work at Google, such as DistBelief [11]. These frameworks have evolved out of dataflow programming paradigms in which the abstractions are operators with input and output

connections. The semantics provided by these languages have been sketched in previous work [5].

TensorFlow employs a dataflow graph of primitive operators extended with restricted control edges to represent differentiable programs. This representation is sufficient for many state-of-the-art models and provides an implementation of reverse mode automatic differentiation [4, 7]. TensorFlow can be viewed as a deeply embedded DSL (eDSL), where the result of executing user's Python script is a computation graph which can then be optimized and transformed before execution. Furthermore, because the graph only exposes high-level nodes, it is possible for the program to be portable to heterogeneous devices, and executing a sub-graph on a given device requires implementation of only those operators for the device. Unfortunately, this programming model has limitations. Because the topology is fixed before execution, TensorFlow does not lend itself well to certain applications. As an example, unmodified TensorFlow does not support building models where the shape of the computation graph is dependent on the input. While there does exist a library to mitigate this particular problem (see [24]), this pattern suggests that should new dependencies become of interest in the future, similar libraries would also have to be written to address each one, entailing considerable engineering effort.

Dynamic frameworks such as Chainer [34], PyTorch [28], Gluon, and TensorFlow eager-mode [33] alleviate this problem by moving from the define-then-run model to the define-by-run model. PyTorch embeds primitives in Python that construct dynamic dataflow graphs. Control flow is executed in the Python interpreter and the dataflow is executed by the framework code with is implemented as Python extension. However when using dynamic frameworks information about control flow is lost, reducing the ability to optimize them. Additionally, dynamic frameworks need to re-optimize any time the graph topology changes, costing CPU cycles and the overhead of moving data between the host and accelerators. This can be solved by transforming the Python code but is effectively the same as a static framework where Python is the input IR.

Previous work in higher-order differentiation is relevant and has informed the Relay design. In particular we have drawn inspiration from various implementations of automatic differentiation [1, 2, 7, 13, 20, 29, 36]. In particular we are interested in techniques that can compute higher order gradients of higher order programs.

Our work is part of the TVM stack [9], which is focused on compiling efficient kernel implementations for deep learning frameworks such as MxNet.

Recent research on the TVM stack [9] has been focused on producing efficient operators (i.e., dense linear algebra kernels), such as generalized matrix multiplication (GEMM) or convolutions. This line of research has focused on low-level performance, but demonstrated the need to tune the high-level computation graph, the operators, and accelerators in tandem to achieve the best performance. High-level transformations on the input program are especially important for the tensorization problem. Tensorization is the analogous process to vectorization in existing compilers, and involves the optimizer decomposing and matching programs to the underlying hardware tensor operations exposed. This problem is more challenging due to being multi-dimensional, mixed size, and non-finite, unlike the analogous SIMD primitives.

The TVM stack is designed to enable a series of fundamental optimizations:

- High-level optimizations, such as operator fusion and layout change
- Memory reuse at the graph and operators level
- Tensorized computations
- Latency hiding (traditional hardware provides this abstraction, but new accelerators push this burden to the compiler writers)

There are multiple related engineering efforts, the primary ones being from Google and Facebook. Facebook has been building an efficient ML stack composed of many projects including Tensor Comprehensions [35] and Glow [31]. Tensor Comprehensions are positioned in a similar space as TVM , but employs different techniques, such as using polyhedral compilation rather than algorithmic schedules. The Glow compiler [31] is similar to NNVM and intended to be a compiler for high-level computation graphs. Glow's design is closer to existing computation graphs, does not appear to be a full language, and is less focused on full-stack tuning.

TensorFlow's XLA is very similar to the complete TVM stack and is focused on providing a lower-level intermediate representation for TensorFlow's computation graph. Relay is designed to replace the user-visible graph with a higher-level abstraction and make it possible for users to write frameworks like TensorFlow and PyTorch in pure Python.

3 Language

Relay is a statically typed, purely functional, differentiable IR. Relay is not a low-level IR intended for writing and optimizing high-performance kernels; rather, it is intended to replace NNVM's computation graph as the input layer of NNVM. We allow for primitive operators implemented either in external languages such as C or C++ or in lower-level IRs like TVM or Tensor Comprehensions. Because Relay is intended as the top layer of the TVM stack [9], we have tight integration with TVM and use it to implement and optimize kernels.

Our intent is for our new IR to serve as a convenient means for researchers to implement new differentiable programming languages and deep probabilistic programming languages in the style of Edward and Pyro.

As we discussed in Section 2, most popular machine learning frameworks construct computation graphs that represent

the user's program. Since these graphs are essentially a modified form of an abstract syntax tree (AST), we consider the transformations and analyses that have been performed on computation graphs as program transforms and program analyses. While other DL frameworks also adopt this perspective, their graph-based approaches have made it difficult to bring the full arsenal of traditional compiler and programming languages techniques to bear.

Static typing enables direct compilation of models into embedded hardware and accelerators, which has been demonstrated in prior work done in the TVM stack [9]. Having an IR like Relay enables the deployment of richer dynamic models for applications such as natural language processing. By taking this point of view, we can leverage decades of programming language research to help us express and understand these deep learning models not as a restricted data flow language, but as a full programming language.

3.1 Grammar and Design

The grammar for the full language can be found in Figure 3.

Relay is a functional language with closures, recursion, conditionals, operators, and tensors. Relay's IR has two main design contributions over computation graphs: the addition of functions and a rich type system that can capture the relationship of tensor operations.

In order to support higher-order (in the sense of higher-order functions) differentiable programs, we need to be able to support computing gradients over arbitrary functions. We accomplish this by introducing a higher-order, higher-order (in both senses) reverse mode operator [29]. This operator allows us to compute nth-order derivatives of higher order programs, opening up the ability to differentiate over arbitrary control structures encoded with functions.

Inspired in part by DLVM [37], a neural network DSL that supports a CFG-style IR for deep learning programs which introduces a type system for tensors that is based on constant tensor shape and types, Relay supports a rich type system that includes dependent typing for tensor shapes, thereby allowing function type signatures to specify the relationship between arguments (such as attributes or other tensors) and the resulting tensor shapes.

4 System Design

NNVM currently represents DL programs as static computation graphs containing operators and input/output data flow. The topology of this graph is fixed, allowing straightforward compilation to TVM 's graph runtime.

We first constructed a prototype in Python to validate our ideas, and to experiment with transformations, such as partial evaluation and automatic differentiation.

Relay is composed of a series of interoperating essential modules:

- A Python frontend, which translates Python code into Relay's C++ data structures.
- A module for automatic differentiation of Relay programs.
- A shape-dependent tensor type system.
- A simple evaluator for prototyping and debugging.
- A type-specialized operator compiler built on TVM.
- An efficient runtime system, which is still in progress.

Below, we describe the design and implementation of the modules that have been prototyped and discuss the in-progress and yet-to-be-implemented components in 5.

4.1 Frontend

Relay currently has two interfaces: a textual AST that can be written in Python or C++ and a Python frontend. We intend to add a JSON serialization interface to allow for easy integration with other compilers.

The Python frontend is the intended user-facing interaction mode for Relay while the other interfaces allow programmatic use of Relay's AST.

The Python interface comprises two pieces: a library and a pair of decorators. The library contains standard DL operators and some Relay-specific functions. The pair of decorators transforms a subset of vanilla Python code into the Relay textual AST representation and generates a wrapper function which will execute that code using one of Relay's evaluation mechanisms.

Although the core of Relay is written in C++, we are able to expose the internals of the system to Python by reusing TVM 's node system, which allows low-effort interoperability between the two languages. We can expose C++ classes in Python simply by inheriting from a special class and writing a class stub in Python.

The Python frontend is inspired by many other projects, which use similar mechanisms to rewrite Python ASTs, such as Tangent [38] [8].

Targeting Python has significant advantages since it has become the lingua franca of the DL community, which is accustomed to Python libraries such as TensorFlow, PyTorch, and Keras. Using Python as a source language also allows users to write and extend Relay in the same language they use to do data processing and deployment.

Figure 4 demonstrates how to use the decorators, and we will briefly outline their semantics below.

Let us preface our description of the decorators by noting that not all of the functionality in this example is currently implemented and this example instead represents the design and ideal syntax for our frontend.

To illustrate the decorators, we briefly trace how the frontend transforms the program in 4 into Relay. In this program, three Python functions have been decorated:

- `lenet`: The declaration of the LeNet model [23].
- `loss`: The loss function of the model.

⟨*Item*⟩ ::= ⟨*Operator*⟩
| ⟨*Definition*⟩

⟨*Operator*⟩ ::= operator ⟨*GlobalId*⟩ : ⟨*Type*⟩

⟨*Definition*⟩ ::= def ⟨*GlobalId*⟩ ((⟨*LocalId*⟩ : ⟨*Type*⟩))* ->
⟨*Type*⟩ { ⟨*Expr*⟩ }

⟨*Expr*⟩ ::= ⟨*LocalId*⟩
| ⟨*GlobalId*⟩
| ℝ
| True
| False
| ⟨*Expr*⟩ ((⟨*Expr*⟩ (, ⟨*Expr*⟩)*)?)
| let ⟨*LocalId*⟩ : ⟨*Type*⟩ = ⟨*Expr*⟩ in ⟨*Expr*⟩
| (⟨*Type*⟩) ⟨*Expr*⟩
| ⟨*Expr*⟩ ⟨*BinOp*⟩ ⟨*Expr*⟩
| ⟨*UnaryOp*⟩ ⟨*Expr*⟩
| ((⟨*Expr*⟩ (, ⟨*Expr*⟩)*)?)
| ⟨*Expr*⟩ [ℕ]
| [⟨*Expr*⟩ (, ⟨*Expr*⟩)*]
| if ⟨*Expr*⟩ then ⟨*Expr*⟩ else ⟨*Expr*⟩
| Zero ⟨*Type*⟩
| Grad ⟨*Expr*⟩
| Ref ⟨*Expr*⟩
| ! ⟨*Expr*⟩
| ⟨*Expr*⟩ := ⟨*Expr*⟩

⟨*BinOp*⟩ ::= +
| -
| *
| /
| !=
| =
| <
| <=
| >
| >=

⟨*UnaryOp*⟩ ::= -
| sq

⟨*Type*⟩ ::= ⟨*BaseType*⟩
| ⟨*Shape*⟩
| Tensor (⟨*Type*⟩ , ⟨*Type*⟩)
| ⟨*Type*⟩ -> ⟨*Type*⟩
| ⟨*TypeId*⟩
| forall (⟨*TypeId*⟩ : ⟨*Kind*⟩) , ⟨*Type*⟩
| RefType (⟨*Type*⟩)
| ((⟨*Type*⟩ (, ⟨*Type*⟩)*)?)

⟨*BaseType*⟩ ::= IntType (ℕ)
| UIntType (ℕ)
| FloatType (ℕ)
| BoolType

⟨*Shape*⟩ ::= Shape ((ℕ (, ℕ)*)?)

⟨*Kind*⟩ ::= BaseType
| Shape
| Type

Figure 3. The BNF Grammar for the Relay langauge. Each case matches a node in our abstract syntax tree. References and related operations cannot be included in frontend user code and are only generated by the reverse-mode automatic differentiation.

- `train_lenet`: The training loop.

Then we have raw Python code at the bottom for facilitating training and inference.

Each parameter of the function requires an explicit type annotation, but the type of local variable assignments can be left out and later be inferred by the back-end.

Any function call in the `relay` namespace is converted to an intrinsic identifier, which must be implemented outside of Relay and registered with the runtime. TVM is the preferred mechanism for implementing them.

In order to prevent the passing of model parameters to every function that needs them, we have two separate decorators: `relay` and `relay_model`. The `relay` decorator declares a function that can be run without any hidden state (and thus, no functions that do require hidden state can be called). The `relay_model` decorator declares a function that cannot be run by default and instead must first be instantiated by a call to `relay.create_model`. When a model is created for a `relay_model`-decorated function, the function's body is searched for any calls that require hidden parameters; any

```
@relay_model
def lenet(x: Tensor[Float, (1, 28, 28)]) -> Tensor[Float, 10]:
    conv1 = relay.conv2d(x, num_filter=20, ksize=[1, 5, 5, 1], no_bias=False)
    tanh1 = relay.tanh(conv1)
    pool1 = relay.max_pool(tanh1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1])
    conv2 = relay.conv2d(pool1, num_filter=50, ksize=[1, 5, 5, 1], no_bias=False)
    tanh2 = relay.tanh(conv2)
    pool2 = relay.max_pool(tanh2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1])
    flatten = relay.flatten_layer(pool2)
    fc1 = relay.linear(flatten, num_hidden=500)
    tanh3 = relay.tanh(fc1)
    return relay.linear(tanh3, num_hidden=10)

@relay
def loss(x: Tensor[Float, (1, 28, 28)], y: Tensor[Float, 10]) -> Float:
    return relay.softmax_cross_entropy(lenet(x), y)

@relay
def train_lenet(training_data: Tensor[Float, (60000, 1, 28, 28)]) -> Model:
    model = relay.create_model(lenet)
    for x, y in data:
        model_grad = relay.grad(model, loss, (x, y))
        relay.update_model_params(model, model_grad)
    return relay.export_model(model)

training_data, test_data = relay.datasets.mnist()
model = train_lenet(training_data)
print(relay.argmax(model(test_data[0])))
```

Figure 4. An example of the Relay Python decorator, which transforms a decorated function into an analogous one in Relay. The defined model is based on LeNet [23] and is trained and tested on the MNIST dataset.

parameters for these calls are then initialized. Note that multiple calls to the same function will still generate multiple sets of hidden parameters. For example, in the `lenet` function, `conv1` and `conv2` both have their own hidden parameters. Initialization for all model parameters is currently assumed to be Gaussian with $\mu = 0$ and some small σ.

To train the model, we define the `loss` function in terms of our model (i.e., `lenet`), and in our training loop, we use `relay.grad` to calculate the gradients of the parameters with respect to the output. Then we pipe the resulting gradients into `relay.update_model_params` to update our parameters (this example uses vanilla stochastic gradient descent). While the Relay IR in general is functional, for convenience, we expose `relay.update_model_params` as a limited form of mutation.

When training is finished, `relay.export_model` returns a callable version of the trained model that can then be used in raw Python.

4.2 Automatic Differentiation

In [29], the authors demonstrate that reverse-mode automatic differentiation can be performed in a functional language by using a local program transformation that introduces references. Our approach is inspired by their insight and is closely related to performing forward-mode automatic differentiation using dual numbers. In the dual number approach, real values are transformed into pairs (called "dual numbers") of the original value and the derivative of the function at that value. All operations in a function are then lifted to operate over dual numbers.

Instead of pairing each value with its partial derivative, as in the forward mode, we pair each real value with a reference of type real, denoting the reverse-mode partial derivative. The reverse-mode partial derivative of a real number is the derivative of that real with respect to the variable representing the final result of the function [26]. Additionally, for every reverse-mode AD transformation we perform, we return a reference to a function from unit to unit, called the "backpropagator." For every real number produced before, the backpropagator is updated to take its partial derivative

and pass it upstream via the references, according to the chain rule. The backpropagator then calls the old version of itself, thus forming a chain of closures to update every partial derivative. For a more detailed explanation, see [29].

We replace each operation over reals with a transformed operation that returns the original value and a zero-initialized reference, then updates the backpropagator to clear the gradient reference, propagate the gradient reference forward, and call the old backpropagator. To phrase it in more AD-specific terms, the Wengert list is constructed dynamically as new reals are created. The Wengert list is represented as the list of closures that created the backpropagator, and the operations to update the list are bundled with the list.

For every generic operation, including control flow and higher-order functions, we only need to transform the inner expression and lift the type to accommodate the new expression. This is identical to what is done in the traditional dual number approach.

Additionally, we extend the syntax with a gradient node (`Grad expr`). In the gradient node, `expr` should be a function from a product of reals to real. The transformed expression is a new function that calculates the result of the original function bundled with all the partial derivatives. This node is implemented by transforming the inner AST with our implementation of reverse mode automatic differentiation. For every real-type argument, we pass the original argument bundled with a new zero-initialized reference to the transformed function. We call the backpropagator, extract the value in the passed reference, clear the references, and return the extracted value in a product with the original result.

This transformation requires us to transform every value inside the passed function, so the function must not contain free variables. (This limitation can always be circumvented by lambda-lifting.) Given a Relay program without free variables, the transformation always produces a valid Relay program, meaning it has the closure property. Thus, we have a higher-order reverse mode, even on programs containing closures. We take this approach over that in [29] for three reasons:

1. **Simplicity**: Pearlmutter and Siskind's approach requires reflection on the AST and closure conversion, which means we would need to implement reflection, algebraic data types, and closure conversion in our own language if we were to follow [29].
2. **Typing**: Additionally, the backpropagators generated in [29] have types that depend on the free variables inside closures. This means the types of the backpropagators are dynamic, which would complicate our type system.
3. **Efficiency**: Reflection and traversing the AST are not fast. While Pearlmutter and Siskind propose to use partial evaluation to remove this overhead, it introduces another layer of complexity.

Currently, we maintain the purity of Relay by only exposing the `Grad` operation. User code can never interact with the references that are produced in the above-described process; the process completely abstracts away the references and returns only the resulting values. We could also potentially make the code produced by the transformation pure by typing it as lazy functional state threads (monads), as presented in [22].

The implementation of automatic differentiation in Relay comprises 449 lines of C++ out of a total of approximately 10 thousand lines in the C++ backend.

4.3 A Type System

Our type system is informed by the authors' previous experience using and implementing dependent type theory. We have kept the language of types small, inspired by type system designs which use small core languages [12, 18].

Our type system allows shape dependency. That is, it allows types to be polymorphic over shapes which can appear both in expressions and types. This design allows us to capture important properties at compile-time, though it sheds the complexity of a traditional dependent type system. Importantly, we have kinding rules which enforce that shapes and base types are both of a different kind from types of values— namely tensors, products, and arrow types.

In this paradigm, knowing all values are tensors allows compiler writers to design and implement optimizations over the AST in a uniform manner. For example, if a user of Relay wants to write an optimization that lifts a computation up one dimension, they can uniformly add a dimension without needing to handle scalar cases. This is very useful for optimizations that change dimension (e.g., auto-batching, spatial packing, or layout changes). We discuss possible extensions to the type system in 5.

The decision to incorporate tensor shape into the type system, rather than to have it as a separate "analysis," allows shape information to be easily stored and reasoned about at any stage of the optimization pipeline and makes it easier for users to be explicit about tensor shapes and their desired effect.

5 Future Work

Relay generalizes NNVM 's computation graph by moving from a limited dataflow language to a full programming language. Relay is intended to act as the top layer of the TVM stack and serves as its input format. Here we detail near-term future work.

$$\frac{width \in \mathbb{N}}{\begin{array}{c}\Delta \vdash \texttt{IntType}(width) : \texttt{BaseType} \\ \Delta \vdash \texttt{FloatType}(width) : \texttt{BaseType} \\ \Delta \vdash \texttt{UIntType}(width) : \texttt{BaseType} \\ \Delta \vdash \texttt{BoolType} : \texttt{BaseType}\end{array}} \text{(BaseType-T)}$$

$$\frac{d_1, d_2, \ldots, d_n \in \mathbb{N}}{\Delta \vdash \texttt{Shape}(d_1, d_2, \ldots, d_n) : \texttt{Shape}} \text{(Shape-T)}$$

$$\frac{\Delta \vdash bt : \texttt{BaseType} \qquad \Delta \vdash sh : \texttt{Shape}}{\Delta \vdash \texttt{Tensor}(bt, sh) : \texttt{Type}} \text{(Tensor-T)}$$

$$\frac{\Delta \vdash T : \texttt{Type} \qquad \Delta \vdash U : \texttt{Type}}{\Delta \vdash T \rightarrow U : \texttt{Type}} \text{(Arrow-T)}$$

$$\frac{K \in \{\texttt{Shape}, \texttt{Type}, \texttt{BaseType}\} \quad \Delta, T : K \vdash body : \texttt{Type}}{\Delta \vdash \texttt{forall}\ (T : K),\ body : \texttt{Type}} \text{(Quantifier-T)}$$

$$\frac{\begin{array}{c}\Delta \vdash T_1 : \texttt{Type} \\ \Delta \vdash T_2 : \texttt{Type} \\ \ldots \\ \Delta \vdash T_n : \texttt{Type}\end{array}}{\Delta \vdash (T_1 \times T_2 \times \cdots \times T_n) : \texttt{Type}} \text{(Product-T)}$$

$$\frac{\Delta \vdash T : \texttt{Type}}{\Delta \vdash \texttt{RefType}(T) : \texttt{Type}} \text{(Ref-T)}$$

Figure 5. Rules for constructing types, indicating kinds. Reference types are only generated internally by reverse-mode automatic differentiation and cannot be given in frontend user code. Also note we will eventually define a more complex AST for shapes.

5.1 Runtime System

Our current evaluator (an interpreter) is a reference implementation used for differential testing and experimentation. This evaluator is not sufficient for experimental evaluation, and the main thrust of our current work is its efficient counter part. An interesting aspect of this evaluator is its use of TVM as a just-in-time compiler to produce type-specialized tensor operators. The optimized runtime system, which is intended as the primary way to deploy and execute Relay programs, is still under heavy development.

Traditional languages have optimized their execution engines' virtual machines for very specific execution profiles, with long-lived heap allocations, and relatively small stack values. DL workloads have a much different execution profile and often do not execute on traditional CPUs, but rather on special-purpose devices, such as GPUs and accelerators.

There are many questions about the lifetime of values and how to handle in-place updates, allocation, reclamation, and more. The runtime system needs new representations of the call stack for functions, new allocation patterns around scopes, and distinct concepts of identity and allocation.

5.2 Optimizations

Relay is designed to provide a whole-program representation of deep learning programs, allowing us to address problems such as host slicing [3], dynamic networks, change of layout, latency hiding, and parallel and distributed scheduling. We have designed Relay with these goals in mind and to help address the critical optimizations identified in [9].

We envision the ability to add other systems' features as optimization passes over Relay programs, for example implementing auto-batching from DyNet[25], operator fusion as done in the current NNVM framework, or change of layout for Tensors. Auto-batching relies on the ability to know about a set of transformations between unbatched operations and batched operations, inserting the appropriate aggregate instructions such as summing in the correct places. Given type information it is possible to extend certain programs with an extra batch dimension, inserting the appropriate operators to preserve typing and semantics.

5.3 Software Engineering

The previous version of Relay supported both a step debugger and the ability to compile Relay programs to Python for debugging and differential testing against other machine learning frameworks. We used this to test automatic differentiation by compiling Relay programs to Python, using the 'autograd' Python library to compute the gradient, then checking the gradient's results using property-based testing [10].

5.4 Numerical Accuracy

ML workloads have proven exceptionally robust to issues of rounding error [15]. Given this tolerance for low-accuracy arithmetic, we are eager to adapt recent techniques for automatically rewriting numerical code to improve accuracy at the cost of performance (e.g., Herbie [27]), to instead trade off accuracy for improved compute. By adapting tools like Herbie and STOKE [32] to the context of machine learning inference and training, Relay will further support developers striving to maximize compute on platforms built around IEEE-754 floating point arithmetic. Moving forward, we hope to further extend these tools and target specialized numerical

$$\frac{i \in \mathbb{Z}}{\Delta; \Gamma \vdash i : \texttt{Tensor}(\texttt{IntType}(32), \texttt{Shape}())} \text{ (Type-Int-Literal)}$$

$$\frac{f \in \mathbb{R}}{\Delta; \Gamma \vdash f : \texttt{Tensor}(\texttt{FloatType}(32), \texttt{Shape}())} \text{ (Type-Float-Literal)}$$

$$\frac{b \in \{\texttt{True}, \texttt{False}\}}{\Delta; \Gamma \vdash b : \texttt{Tensor}(\texttt{BoolType}, \texttt{Shape}())} \text{ (Type-Bool-Literal)}$$

$$\frac{\begin{array}{c}\Delta \vdash s = \texttt{Shape}(d_1, d_2 \ldots, d_n) \qquad \Delta \vdash b : \texttt{BaseType} \\ \Delta; \Gamma \vdash t_1 : \texttt{Tensor}(b, s) \qquad \Delta; \Gamma \vdash t_2 : \texttt{Tensor}(b, s) \\ \ldots \qquad \Delta; \Gamma \vdash t_m : \texttt{Tensor}(b, s)\end{array}}{\Delta; \Gamma \vdash [t_1, t_2, \ldots, t_m] : \texttt{Tensor}(b, \texttt{Shape}(m, d_1, d_2, \ldots, d_n))} \text{ (Type-Tensor-Literal)}$$

$$\frac{\Delta; \Gamma \vdash p_1 : T_1 \qquad \Delta; \Gamma \vdash p_2 : T_2 \qquad \ldots \qquad \Delta; \Gamma \vdash p_n : T_n}{\Delta; \Gamma \vdash (p_1, p_2, \ldots, p_n) : T_1 \times T_2 \times \cdots \times T_n} \text{ (Type-Product)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash p : T_1 \times T_2 \times \cdots \times T_n \\ i \in [0, n)\end{array}}{\Delta; \Gamma \vdash p[i] : T_i} \text{ (Type-Projection)}$$

$$\frac{\Delta; \Gamma \vdash d : T \qquad \Delta; \Gamma, id : T \vdash b : T'}{\Delta; \Gamma \vdash \texttt{let } id = d \texttt{ in } b : T'} \text{ (Type-Let)}$$

$$\frac{\begin{array}{c}op \in \{\texttt{-}, \texttt{sq}\} \qquad \Delta \vdash b : \texttt{BaseType} \qquad \Delta \vdash s : \texttt{Shape} \\ \Delta; \Gamma \vdash t : \texttt{Tensor}(b, s)\end{array}}{\Delta; \Gamma \vdash \texttt{UnaryOp}(op, t) : \texttt{Tensor}(b, s)} \text{ (Type-UnaryOp)}$$

$$\frac{\begin{array}{c}op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\} \qquad \Delta \vdash b : \texttt{BaseType} \qquad \Delta \vdash s : \texttt{Shape} \\ \Delta; \Gamma \vdash t_1 : \texttt{Tensor}(b, s) \qquad \Delta; \Gamma \vdash t_2 : \texttt{Tensor}(b, s)\end{array}}{\Delta; \Gamma \vdash \texttt{BinaryOp}(op, t_1, t_2) : \texttt{Tensor}(b, s)} \text{ (Type-Noncomp-BinaryOp)}$$

$$\frac{\begin{array}{c}op \in \{\texttt{=}, \texttt{!=}, \texttt{>}, \texttt{<}, \texttt{>=}, \texttt{<=}\} \\ \Delta \vdash b : \texttt{BaseType} \qquad \Delta \vdash s : \texttt{Shape} \\ \Delta; \Gamma \vdash t_1 : \texttt{Tensor}(b, s) \qquad \Delta; \Gamma \vdash t_2 : \texttt{Tensor}(b, s)\end{array}}{\Delta; \Gamma \vdash \texttt{BinaryOp}(op, t_1, t_2) : \texttt{Tensor}(\texttt{BoolType}, s)} \text{ (Type-Comp-BinaryOp)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma, p_1 : T_1, p_2 : T_2, \ldots, p_n : T_n, f : (T_1 \times T_2 \times \cdots \times T_n) \to T' \\ \vdash body : T'\end{array}}{\begin{array}{l}\Delta; \Gamma \vdash \texttt{def } f(p_1 : T_1, p_2 : T_2, \ldots, p_n : T_n) \texttt{ -> } T', body \\ \quad : (T_1 \times T_2 \times \cdots \times T_n) \to T'\end{array}} \text{ (Type-Function-Definition)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash f : (T_1 \times \cdots \times T_n) \to T' \\ \Delta; \Gamma \vdash a_1 : T_1 \qquad \Delta; \Gamma \vdash a_2 : T_2 \qquad \ldots \qquad \Delta; \Gamma \vdash a_n : T_n\end{array}}{\Delta; \Gamma \vdash f(a_1, a_2, \ldots, a_n) : T'} \text{ (Type-Call)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash c : \texttt{Tensor}(\texttt{BoolType}, \texttt{Shape}()) \\ \Delta; \Gamma \vdash b_1 : T \qquad \Delta; \Gamma \vdash b_2 : T\end{array}}{\Delta; \Gamma \vdash \texttt{if } c \texttt{ then } b_1 \texttt{ else } b_2 : T} \text{ (Type-If)}$$

$$\frac{\Delta \vdash b : \texttt{BaseType} \qquad \Delta \vdash s : \texttt{Shape}}{\Delta; \Gamma \vdash \texttt{Zero Tensor}(b, s) : \texttt{Tensor}(b, s)} \text{ (Type-Zero)}$$

$$\frac{\Delta; \Gamma \vdash \texttt{autodiff}(e) : T}{\Delta; \Gamma \vdash \texttt{Grad } e : T} \text{ (Type-Gradient)}$$

$$\frac{\Delta; \Gamma \vdash n : T}{\Delta; \Gamma \vdash \texttt{Ref } n : \texttt{RefType}(T)} \text{ (Type-Ref)}$$

$$\frac{\Delta; \Gamma \vdash r : \texttt{RefType}(T)}{\Delta; \Gamma \vdash !r : T} \text{ (Type-Val-Ref)}$$

$$\frac{\Delta; \Gamma \vdash r : \texttt{RefType}(T) \qquad \Delta; \Gamma \vdash v : T}{\Delta; \Gamma \vdash r := v : ()} \text{ (Type-Set-Ref)}$$

Figure 6. Rules for deriving types of expressions and definitions. The unit type, (), is syntactic sugar for a product type with zero members. Note that these type rules assume that all type variables in quantifiers have already been concretely instantiated. Additionally, in the rule for gradient, "autodiff" is the automatic differentiation AST transformation on expression e; rather than attempt to capture the entire semantics of the transformation in that inference rule, we explain the transformation in 4.2.

representations including mixed width and fixed point computations; blocked floating point; non-standard, accelerator-specific numerics; and emerging alternate standards (e.g., the work on unums and posits [16]).

5.5 Type System Extensions

One planned type system extension is handling tensors with partially-specified shapes, that is, shapes where some dimensions are unknown. This is useful for many NLP applications, where the data may be jagged in one or more dimensions and not representable with a fixed shape.

One other extension is expanding the type system to track individual tensors' data layouts. This is motivated by the

difficulties we have encountered writing change-of-layout optimizations, which both must infer existing layouts and ensure all uses are transformed. These types of errors have led to hard-to-debug code that silently produces incorrect results or crashes. By making these change-of-layout operations explicit, it would be possible to perform optimizations in that style around automatic boxing and unboxing of values.

A more significant extension would be an integrated effect system, allowing us to segregate code manipulating different resources such as random number generators, state, I/O and so on. This kind of change is more radical and for now is left as an analysis that must be performed by the compiler.

6 Conclusion

We describe an in-progress implementation of Relay: a new IR for efficient compilation and execution of machine learning models. We are designing Relay to be the core of the second version of NNVM and to address key challenges both researchers and engineers face using today's computation graphs. Our initial prototype implements our vision of how a researcher might write models in Relay, with the ergonomics of vanilla Python as well as the advantages enjoyed by systems like PyTorch and TensorFlow. Relay's implementation is still in flux, and we are focused on exploring topics in section 5. We believe Relay to be an important part of the TVM stack that will facilitate both current and future research efforts.

Acknowledgments

This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program cosponsored by DARPA. The authors would also like to thank Calvin Loncaric, Pavel Panchekha, Vinod Grover, and Eunice Jun for discussion, insightful comments, and feedback on earlier drafts.

References

[1] [n. d.]. DeepDarkFantasy - A Programming Language For Deep Learning. ([n. d.]). https://github.com/ThoughtWorksInc/DeepDarkFantasy

[2] [n. d.]. DeepLearning.scala. ([n. d.]). https://github.com/ThoughtWorksInc/DeepLearning.scala

[3] [n. d.]. Swift for TensorFlow. ([n. d.]). https://www.tensorflow.org/community/swift

[4] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[5] Martin Abadi, Michael Isard, and Derek G. Murray. 2017. A Computational Model for TensorFlow (An Introduction). http://dl.acm.org/citation.cfm?doid=3088525.3088527

[6] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.01989 (2016). arXiv:1611.01989 http://arxiv.org/abs/1611.01989

[7] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2015. Automatic differentiation in machine learning: a survey. *CoRR* abs/1502.05767 (2015). arXiv:1502.05767 http://arxiv.org/abs/1502.05767

[8] Oliver Breuleux and Bart van MerriÃńnboer. 2017. Automatic differentiation in Myia. (2017). https://openreview.net/pdf?id=S1hcluzAb

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Compilation Stack for Deep Learning. In *SysML 2018*. https://arxiv.org/abs/1802.04799

[10] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[11] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, MarcâĂŹAurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *NIPS*. https://research.google.com/archive/large_deep_networks_nips2012.html

[12] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 34 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110278

[13] Conal Elliott. 2009. Beautiful differentiation. In *International Conference on Functional Programming (ICFP)*. http://conal.net/papers/beautiful-differentiation

[14] Shixiang Gu, Ethan Holly, Timothy P. Lillicrap, and Sergey Levine. 2016. Deep Reinforcement Learning for Robotic Manipulation. *CoRR* abs/1610.00633 (2016). arXiv:1610.00633 http://arxiv.org/abs/1610.00633

[15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. *CoRR* abs/1502.02551 (2015). arXiv:1502.02551 http://arxiv.org/abs/1502.02551

[16] Gustafson and Yonemoto. 2017. Beating Floating Point at Its Own Game: Posit Arithmetic. *Supercomput. Front. Innov.: Int. J.* 4, 2 (June 2017), 71–86. https://doi.org/10.14529/jsfi170206

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[18] Simon Peyton Jones. [n. d.]. Into the Core - Squeezing Haskell into Nine Constructors. ([n. d.]). https://www.youtube.com/watch?v=uR_VzYxvbxg

[19] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox,

and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR* abs/1704.04760 (2017). arXiv:1704.04760 http://arxiv.org/abs/1704.04760

[20] Edward Kmett, Barak Pearlmutter, and Jeffrey Mark Siskind. 2008. ad: Automatic Differentiation. (2008). https://github.com/ekmett/ad

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., USA, 1097–1105. http://dl.acm.org/citation.cfm?id=2999134.2999257

[22] John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/178243.178246

[23] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[24] Moshe Looks, Marcello Herreshoff, and DeLesley Hutchins. 2017. Announcing TensorFlow Fold: Deep Learning With Dynamic Computation Graphs. https://research.googleblog.com/2017/02/announcing-tensorflow-fold-deep.html. (7 February 2017).

[25] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. [n. d.]. DyNet: The Dynamic Neural Network Toolkit. ([n. d.]). https://arxiv.org/abs/1701.03980

[26] Christopher Olah. 2015. Calculus on Computational Graphs: Backpropagation. (2015). http://colah.github.io/posts/2015-08-Backprop/

[27] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, Vol. 50. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

[28] 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). https://openreview.net/pdf?id=BJJsrmfCZ

[29] Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a Functional Framework: Lambda the Ultimate Backpropagator. *ACM Trans. Program. Lang. Syst.* 30, 2, Article 7 (March 2008), 36 pages. https://doi.org/10.1145/1330017.1330018

[30] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[31] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* abs/1805.00907 (2018). arXiv:1805.00907 https://arxiv.org/abs/1805.00907

[32] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/2594291.2594302

[33] Asim Shankar and Wolff Dobson. 2017. Eager Execution: An imperative, define-by-run interface to TensorFlow. (2017). https://ai.googleblog.com/2017/10/eager-execution-imperative-define-by.html

[34] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: a Next-Generation Open Source Framework for Deep Learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*. http://learningsys.org/papers/LearningSys_2015_paper_33.pdf

[35] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. (2018). arXiv:1802.04730 https://arxiv.org/abs/1802.04730

[36] Mu Wang and Alex Pothen. 2017. An Overview of High Order Reverse Mode. (2017). https://openreview.net/pdf?id=Hkmj6tzRZ

[37] Richard Wei, Vikram S. Adve, and Lane Schwartz. 2017. DLVM: A modern compiler infrastructure for deep learning systems. *CoRR* abs/1711.03016 (2017). arXiv:1711.03016 http://arxiv.org/abs/1711.03016

[38] Alex Wiltschko. 2017. Tangent: Source-to-Source Debuggable Derivatives. (2017). https://ai.googleblog.com/2017/11/tangent-source-to-source-debuggable.html