**torch.fx: Practical Program Capture and Transformation for Deep Learning in Python**

James K Reed · Zachary DeVito · Horace He · Ansley Ussery · Jason Ansel

**ABSTRACT**

Modern deep learning frameworks provide imperative, **eager execution** programming interfaces embedded in Python to provide a productive development experience. However, deep learning practitioners sometimes need to capture and transform program structure for performance optimization, visualization, analysis, and hardware integration. We study the different designs for program capture and transformation used in deep learning. By designing for typical deep learning use cases rather than long tail ones, it is possible to create a simpler framework for program capture and transformation. We apply this principle in **torch.fx**, a program capture and transformation library for PyTorch written entirely in Python and optimized for high developer productivity by ML practitioners. We present case studies showing how **torch.fx** enables workflows previously inaccessible in the PyTorch ecosystem.

**1 INTRODUCTION**

Early **graph mode** or **define-and-run** (Tokui et al., 2019) deep learning frameworks like Caffe (Jia et al., 2014), Theano (Al-Rfou et al., 2016), and TensorFlow (Abadi et al., 2016) defined APIs in which the user constructed a graph-based intermediate representation (IR) of the desired computation. Program transformations like program differentiation, device/host partitioning and placement, quantization, device lowering, and performance optimization could be applied directly to this IR. One way to think of these frameworks is as simple embedded programming languages that are meta-programmed from a host language, predominantly Python (Innes et al., 2017).

However, these frameworks require the user to exit the host language and enter a domain-specific language and runtime, which often has inferior user experience compared to the host language. For instance, debugging requires different tools from the typical debugging toolkits such as Python’s pdb library.

More recent **eager mode** or **define-by-run** (Tokui et al., 2019) frameworks such as Autograd (Maclaurin et al., 2015), Chainer (Tokui et al., 2019), PyTorch (Paszke et al., 2019) and TensorFlow Eager (Agrawal et al., 2019) eschew explicit graph-building APIs in favor of programming in the host language directly. The primary program transformation used in deep learning frameworks, program differentiation, is reformulated from an ahead-of-time transformation to a just-in-time transformation, in the form of auto-differentiation.

Most training and inference can be done using eager mode with auto-differentiation. However, there are still transformations—such as program quantization or operator fusion—that are easier to write given the additional program structure provided by an IR. To bridge this gap, an eager-mode framework needs a way of capturing program structure from user programs to enable these transformations.

Some program capture systems are built to capture a free-standing representation of the whole program for the purposes of serialization or export. For instance, TorchScript (DeVito et al., 2018) includes mutable state, control-flow, and complex data types for the purposes of faithfully modeling the semantics of the original Python program. Modeling Python in full generality comes at the cost of complexity in program capture techniques and difficulty of writing transforms on the highly-complex IR.

In contrast, it is possible to decouple the requirements of faithfully modeling Python from the requirements needed for transforms such as quantization or fusion. Transforms are often formulated as modifications to a high-level directed acyclic graph (DAG) organization of the code, with implementation details hidden within high-level blocks (such as Convolution or Batch Normalization). Thus, simplifications can be made to both the program capture mechanism and the IR it produces, focusing on the high-level DAG structure of the majority of neural network computation.

For this use case, we present **torch.fx**, a high-productivity
library for capturing and transforming PyTorch programs. torch.fx explicitly trades generality of supported programs for simplicity of program capture and representation. torch.fx focuses on the DAG representation of deep learning programs and provides customization interfaces to adapt programs into this representation. In doing so, torch.fx is able to provide a program transform interface that supports the majority of deep learning programs while providing simple and easy-to-use APIs for implementing transforms.

We present the following contributions:

1. A practical analysis of the features of program capture and transformation that are important for deep learning programs.
2. A Python-only program capture library that implements these features and can be customized to capture different levels of program detail.
3. A simple 6 instruction IR for representing captured programs that focuses on ease of understanding and ease of doing static analysis.
4. A code generation system for returning transformed code back to the host language’s ecosystem.
5. Case studies in how torch.fx has been used in practice to develop features for performance optimization, program analysis, device lowering, and more.

2 BACKGROUND

When capturing and transforming programs, both eager and graph-mode frameworks must make choices about capturing program structure, program specialization and the design of the intermediate representation in which programs are kept. The combination of these choices determines the space of programs that are representable in the framework, the ease of writing transformations, and the performance of resulting transformed programs. In general, supporting more programs at high performance requires a more complicated capture framework and IR and subsequently makes transformations harder to write.

2.1 Capturing Program Structure

There are several ways to capture program structure from Python programs. The simplest way is to trace the execution of a model given some example inputs and record the operations that occur, which is the approach used by PyTorch’s jit.trace (DeVito et al., 2018). A slightly more complicated variant of this approach is to perform tracing with abstract values rather than example inputs (symbolic tracing). MXNet’s Gluon (Chen et al., 2015), and TensorFlow’s tf.function (Moldovan et al., 2018) implement this approach. In addition to the user not having to provide example inputs, this approach surfaces locations where Python control flow depends on the input values, rather than collecting a trace specialized to the control decisions imparted by the example inputs.

During tracing, operations are only recorded for tensors and a small number of other data structures such as lists of tensors. This means that tracing can only record a representation for a subset of the Python program. Although tracing’s visibility into the program is limited, this is often sufficient for deep learning computations, which are most often flat sequences of tensor operations—termed basic block programs in Section 2.3.

By overriding the execution behavior of standard Python code, some tracing systems can capture more program structure, such as control flow, at the cost of additional complexity. For instance, tf.function augments symbolic tracing with a Lightweight Modular Staging (Rompf & Odersky, 2010) system that uses Python AST transforms to convert imperative control flow constructs into higher-order Python functions, which can then be traced.

An alternative way to capture program structure is to have users write models directly in an embedded programming language within Python. The simplest of these techniques is to provide a graph-building API similar to TensorFlow, which lets users build programs (graphs) by calling Python functions. It is awkward to represent control flow in these APIs, so PyTorch’s TorchScript (DeVito et al., 2018) instead extracts programs directly from the Python source using a traditional lexer-parser-compiler toolchain. TorchScript can inspect the source syntax in full fidelity and can understand language constructs such as structured control flow, collection types (e.g. tuple, list, dict) and user-defined types. As opposed to tracing, which can fail silently, embedded language approaches can report unsupported constructs as part of compilation. On the other hand, embedded language compilation is significantly more complicated to implement, since it requires a full language stack. Even then, in practice these systems will not support the full Python language, so users still need to make their program conform to the supported subset (albeit a larger subset than supported by tracing systems).

Systems such as Zygote.jl (Innes, 2018) and TPU integration (Fischer & Saba, 2018) in the Julia ecosystem (Bezanson et al., 2017) as well as Swift for TensorFlow (Saeta et al., 2021) provide program transformation interfaces by way of integration into non-Python host languages. The main drawback of such native host language integrations in Swift and Julia is that they require the user to exit the Python ecosystem. Python has considerable momentum and extensive libraries in the numeric/scientific computing (and particularly deep learning) space, and many users prefer
to stay in the Python ecosystem. While other languages may provide objectively better experiences in some respects, adoption has been slow.

2.2 Specializing Programs

A Python expression such as `a + b` is very abstract. There are no constraints on the types of `a` or `b`. Even if both are Tensors, the number of dimensions and the size of the dimensions might vary. When ML frameworks capture programs, they often simultaneously specialize these expressions such that they are only valid for specific types or tensor shapes. The more a program is specialized, the fewer inputs it will work on, so approaches vary in the degree of specialization, the timing of when specialization is done (ahead of time, just-in-time), and the safety of the specialized result.

For example, PyTorch’s TorchScript `torch.jit.trace` (DeVito et al., 2018) specializes to the shape of the example inputs. `jit.trace` capture is unintrusive—that is—it records the operations that occur during an actual execution run of the program. One implication of this is the presence of tensor metadata such as the `ndim` or shape attributes, which can escape the traced region and be used in control decisions within the Python program. This may cause the traced representation to be shape specialized—that is—it is only valid for the value shapes used at runtime and may fail for other shapes.

To avoid the problem of specialization failing for some inputs, systems such as DyNet (Neubig et al., 2017) and LazyTensor (Suhan et al., 2021) perform tracing just-in-time, and thus can capture specialized program representations for every invocation. At runtime, these systems defer execution of tensor operations, instead accumulating a program trace. When a value must be materialized, the system will apply transformations to the collected program representation (e.g. automatic batching or native code lowering) and execute the code, returning the values requested. However, this process adds additional cost, since the program is captured on every invocation. LazyTensor uses a caching system to reduce this cost: optimized artifacts are stored in a cache keyed by a hash of the collected IR. Further invocations of the same IR, the optimized artifact can be called directly.

The performance of JIT specialization can also be improved by proving that re-capturing the program is unneeded for some inputs. For instance, JAX’s `jit.combinator` (Frostig et al., 2018) uses pure, functional Python programs as input. This enforces referential transparency on non-Tensor computation like shape expressions. When some transform requires specialization, such as conversion to XLA (The XLA Team, 2017) with static shapes, the system can look at the shapes of the inputs to determine if a new capture is required. A disadvantage of JIT specialization is that it is more complicated to reason about code execution. For instance, `print` or `pdb` statements in traced code will only be executed on runs where re-tracing occurs. Re-tracing and re-transformation can also cause hard-to-predict performance bubbles as execution of the system stalls to re-specialize.

2.3 Intermediate Representation Design

ML frameworks vary in the format of their IRs, with richer IRs capturing more programs and being more expressive at the cost of additional complexity to write transformations or run the code efficiently.

Language Many frameworks define their IR in a cross-language way. For example, Caffe and TensorFlow use the Protocol Buffers format (Xiao et al., 2008) to represent computational graphs. PyTorch’s JIT and MXNet use C++ data structures for their IR with additional bindings into Python. Such native representations can have better runtime performance and may be easier to serialize. On the other hand, these representations can impose a learning curve above that required for programming Python.

Control flow Most neural networks are expressible as flat sequences of tensor operations without control flow such as if-statements or loops—a definition we refer to as a basic block program. Basic block programs are often represented as a directed acyclic graph (DAG) data structure. Multilayer Perceptrons (MLPs), Convolutional Neural Networks (CNNs) such as ResNet (He et al., 2015) and personalization/recommendation models (Naumov et al., 2019) are easily expressed this way. Similarly, Transformer networks (Vaswani et al., 2017) can also be expressed in this way, barring the loop needed for sequence generation on the decoder portion of the network.

Recurrent Neural Networks (RNNs) such as the Elman RNN (Elman, 1990), LSTM (Hochreiter & Schmidhuber, 1997), and Gated Recurrent Unit (GRU) (Cho et al., 2014) are not immediately expressible in this way, as the recurrent network computation is applied repeatedly across elements of a sequence with (typically) dynamic length. RNN structures can be represented in an imperative language as a loop with tensor computation applied in the loop body and tensor values carried across loop iterations. However, in practice, these RNN structures are typically provided as wholesale tensor operations. Thus, an entire RNN application over a sequence appears in code as a call to an RNN function or module. Therefore, these network architectures often also appear as basic block programs.

Nevertheless, many frameworks support capturing and representing control flow in their IR. TorchScript built control flow support into all of its components first-class due to anticipation for workloads to become more complex, particularly in sequence processing domains. JAX uses higher-
order functions such as \texttt{jax.lax.scan} to allow functional-

style control flow (Frostig et al., 2018). MLIR represents

control flow with basic blocks that end in tail calls (Lattner

et al., 2020). In addition to adding complexity to the IR,

more general control flow also makes transforms such as

common sub-expressions more complicated to implement.

\textbf{State} Deep learning models contain state in the form of

the trainable model weights used in different layers. Apart

from these parameters, most networks operate as pure func-

tions of their inputs. ML frameworks take different ap-

proaches to handling how this state is mutated.

PyTorch allows values to be mutated and tensors can be

views of each other. For example, the slicing syntax \texttt{x[i]}

(where \(x\) is a Tensor value) does not produce a new Tensor

value, but rather returns a view aliasing the subset of tensor

\(x\) indexed by \(i\). Views can also be mutated. For example,

the expression \(x[i] = y\) will write the value of \(y\) into the

portion of \(x\) indexed by \(i\).

Since PyTorch supports these aliasing and mutation seman-
tics, modifications to programs must be done in the context

of an analysis that proves that the modification is safe (An-
dersen, 1994). TorchScript implemented such alias analysis

for the purpose of reasoning about the safety of transforms

over the TorchScript IR. However, this comes at a high cost:

all operations in the program must be annotated with infor-

mation specifying their aliasing and mutation behavior. In

practice, many functions (opaque calls or ones that have not

been annotated with relaxed semantics) are treated with a

conservative assumption that the callee mutates global mem-

ory, causing the operation to act as a barrier and hindering

optimization. Needing to reason about aliasing and mutabil-

ity complicates pass authoring, adds additional maintenance

burden to the framework, and can limit optimization oppor-
tunities, but enables the user to apply the full generality of

the PyTorch tensor language.

JAX’s functional approach moves the burden of tracking this

state outside of the framework. Instead the model must be

turned into a pure function where the parameters are passed

as inputs. Typically, this is done with wrapper libraries

such as Haiku (Hennigan et al., 2020) or Flax (Heek et al.,

2020). Any transforms that have to modify both state and
code, such as folding batch norm scaling to a weight tensor,
are made more complicated because these components no
longer live together in the same framework.

\section{Design Principles}

Many of the different designs for program capture and trans-

formation used in existing frameworks favor the ability to

represent more deep learning programs at the cost of the

complexity of their implementation. When captured pro-

grams are the only way to run a program, the ability to

capture a program in full fidelity is crucial. But PyTorch is

primarily used as an \textit{eager execution} framework and pro-
cram capture is only used for some specific transforms; It

does not need to work for an entire program. Furthermore,

most PyTorch programmers who want to transform models

are machine learning practitioners who prefer to work in

Python and may have less knowledge of compiler design.

By designing for typical deep learning models rather than

the long tail, it is possible to create a framework that is much

easier to use and simpler to implement. This philosophy is

captured by 	exttt{torch.fx}’s design principles:

- Prefer making program capture and transformation easy

  for typical models at the cost of working for all

  possible programs. Avoid complexity to support long-

  tail, esoteric use cases.

- Work with tools and concepts that ML practitioners are

  already familiar with such as Python data structures

  and the publicly documented operators in PyTorch.

- Make the process of program capture highly configurable

  so users can implement their own solutions for

  long-tail uses. Allowing users to make one-off config-

  urations is simpler than handling the general case.

\section{TORCH.FX Overview}

In the spirit of simplicity, \texttt{torch.fx captures programs}

via symbolic tracing, \textit{represents them} using a simple 6-

instruction python-based IR, and \textit{re-generates Python code}

from the IR to execute it. To avoid the complexities of re-
capture for JIT specialization, \texttt{torch.fx} makes no attempt to

specialize programs itself, instead relying on the transforms

to decide what specializations they want to perform during

capture. The process of symbolic tracing can be configured

by users to work for more esoteric uses.

Figure 1 shows an example of capturing code with \texttt{torch.fx}.

\texttt{symbolic_trace} takes a function or \texttt{torch.nn.Module} and

captures its structure in a \texttt{Graph} object. That \texttt{Graph} object

is combined with module parameters in a \texttt{GraphModule}, which

is a subclass of \texttt{torch.nn.Module} whose forward method

runs the captured \texttt{Graph}. We can print the \texttt{Nodes} of this

\texttt{Graph} to see the IR that was captured. \texttt{placeholder} nodes

represent inputs and a single \texttt{output} node represents the

result of the \texttt{Graph}. \texttt{call_function} nodes have a reference

directly to the Python function they would call. \texttt{call_method}

nodes directly invoke a method on their first argument. The

\texttt{Graph} is reconstituted into Python code (\texttt{traced.code}) for

invocation.

Figure 2 shows an example transform using \texttt{torch.fx}. The

transform finds all instances of one activation and replaces
import torch
from torch.fx import symbolic_trace, GraphModule

def my_func(x):
    return torch.relu(x).neg()

# Program capture via symbolic tracing
traced : GraphModule = symbolic_trace(my_func)
for n in traced.graph.nodes:
    print(f'{n.name} = {n.op} target={n.target} args={n.args}

x = placeholder target=x args=()
relu = call_function target=\_\_call\_\_method\_\_relu \_\_\_ args=(x,)
eg = call_method target=neg args=(\_\_call\_\_\_relu,)
output = output target=\_\_output\_\_ args=(\_\_call\_\_\_neg,)

print(traced.code)

# Changes in the new program

def forward(self, x):
    relu = torch.relu(x); x = None
    neg = relu.neg(); relu = None
    output = output + x
    return output

traced.recompile()

Figure 1. torch.fx captures programs using symbolic tracing into a simple IR and generates Python code from that IR.

from torch.fx import Graph

def replace_activation(g: Graph, old, new):
    for n in g.nodes:
        if n.op == 'call_function' and n.target == old:
            # create IR to call new activate
            with g.inserting_after(n):
                new_n = g.call_function(new, n.args)
                n.replace_all_uses_with(new_n)
                g.erase_node(n)
            # or for this simplified case: `n.target = new'
replace_activation(traced.graph, torch.relu,
                   torch.nn.functional.gelu)
traced.recompile()

Figure 2. Transforms, like this one that replaces activation functions, are written directly in Python.

4.1 Program Capture

torch.fx’s symbolic tracing mechanism uses a Proxy data structure to record operations on values flowing through the program. Proxy is a duck-typed Python class that records attribute accesses and method calls on it, acting as an abstract value that stands in for the concrete program values. Proxy uses the __torch_function__ protocol (Abbasi et al., 2020) to intercept and record the dispatch of PyTorch operators, which are free functions. Finally, torch.fx overrides PyTorch’s Module abstraction to record calls to Modules using proxied values. The process of symbolic tracing is configurable via a Tracer class whose methods can be overridden to control what values are kept as Proxys and which are partially evaluated during the trace.

4.2 Intermediate Representation

torch.fx represents programs in a DAG-based IR, which is amenable to the basic block programs common in deep learning. Programs are represented as a Graph object, which contains a linear series of Node objects representing operations. Nodes have a string opcode, describing what type of operation the Node represents (the semantics of the opcodes can be found in Appendix A.1). Nodes have an associated target, which is the call target for call nodes (call_module, call_function, and call_method). Finally, Nodes have args and kwargs, which together represent the arguments to the target in the Python calling convention as witnessed during tracing1 (the semantics for args and kwargs for each opcode can be found in Appendix A.2). Data dependencies between Nodes are represented as references to other Nodes within args and kwargs.

To simplify the IR, torch.fx’s IR does not have primitive operations that model the construction or mutation of data structures. Nevertheless, args and kwargs support immediate values: Python built-in types such as int and float and recursive collection types like tuple and list can appear as Node arguments without separate object construction Nodes. Because Nodes support immediate values, the IR is clean and Nodes are approximately 1-to-1 with Tensor operations.

torch.fx stores the state of the program in the GraphModule class. GraphModule is the container for transformed programs, exposing the transformed, generated code as well as providing the familiar parameter management APIs of nn.Module. GraphModule can be used anywhere a normal nn.Module can be used, providing interoperability between transformed code and the rest of the PyTorch ecosystem.

torch.fx’s IR provides two opcodes for accessing state in the Module hierarchy: call_module, which invokes a sub-Module’s forward method, and get_attr, which fetches a parameter from the Module. Transformed code can interact with the Module hierarchy in much the same way normal PyTorch code can via these opcodes. In addition, transformations can manipulate the mutable state in the Module hierarchy simultaneously with transformations over code. This provides a natural separation between the mutable parameters and the functional Graph that interacts with them via call_module Nodes, while still keeping them together in a single object for doing transformations that work on both.

4.3 Source-to-Source Transformation

The final stage in the torch.fx transformation pipeline is code generation. Rather than exiting the Python ecosystem and entering a bespoke runtime, torch.fx generates

1No normalization is applied to args or kwargs; They are preserved as the user wrote them. This facilitates further backward-compatibility of the generated code.
torch.fx: Practical Program Capture and Transformation for Deep Learning in Python

5.2 Configurable Program Capture

torch.fx’s symbolic tracing is customizable. A Tracer class controls the behavior of fx.symbolic_trace. Its methods can be overridden to change the tracing process’s behavior.

The is_leaf_module method can be overridden to specify which PyTorch Module instances should be treated as opaque calls during tracing. By default, torch.fx keeps PyTorch built-in Modules such as nn.Conv2d intact while tracing through user-defined Modules, since this creates a trace of standard, understandable primitives. Customizing this behavior can block out portions of a model that contain unsupported language features or modify the level of representation used for transformations.

create_proxy is a method that can be overridden to customize the behavior of creating a Node in the Graph and the associated runtime Proxy value. This can be used to, for example, install custom metadata onto Nodes for the purpose of transformation or to support custom data structures as traceable values. A custom Tracer could, for instance, specialize the sizes and shapes of Tensors and use these values to capture a program that would otherwise not be traceable without specialization.

5.3 AoT Capture without Specialization

While ahead-of-time tracing limits the space of programs that can be captured (e.g. arbitrary control flow is not supported), it provides a more predictable and more observable capture, transformation, and code generation process that fits into the PyTorch developer experience and works well in practice.

Unlike example-based tracing, symbolic tracing cannot incidentally specialize program flow because the information needed to make data-dependent control flow decisions is not present at trace time. Common Tensor attributes used in control decisions such as shape and ndim are returned as Proxy values during symbolic tracing. Operations on these values can then be recorded. On the other hand, when these Proxy objects are used in a context where untraceable operations (such as a cast to Python built-in types like int or bool) occur on them, the user receives an error message describing the problem and a stack trace indicating the location of the issue.

5.4 Python-based IR and Transforms

Rather than use a cross-language format such as protocol buffers, torch.fx IR is entirely represented and implemented Python. Users can call, read, or override it easily. There is no need to understand Protocol Buffers or C++ (or set up either of their build environments), which present barriers to ML engineers familiar with working primarily in
With Transformers (Vaswani et al., 2017) increasingly replacing sequential recursive neural networks with larger scalable attention modules, the use of host language control flow in deep learning is becoming more rare. Many models can be expressed without it, and even for programs with some control flow (e.g. a beam search decoder), there are large blocks of the model without control flow (the encoder and the step of the decoder).

However, the presence of control flow in an IR adds significant complexity regardless of whether a particular model uses it. Most analyses on the IR must be expressed as fix-point data-flow (Kildall, 1972) over the program rather than simple forward propagation. The author must define a lattice, transfer function, and join function for the analyzed property in the program and prove monotonicity and finiteness thereof. While familiar to compiler writers, we have found that writers of ML transforms often introduce bugs in transforms such as having join functions that are not monotonic or failing to iterate until converged. In contrast, for a basic block IR, only a transfer function is needed.

An example of the complexity of fix-point analysis can be found in shape propagation: shapes can be trivially propagated forward through a basic block program (barring a few operations with value-dependent output shapes). However, when control flow is added, shape propagation does not satisfy the finiteness property—a value carried across a loop iteration can take on an infinite number of shapes, as shown in Figure 4. The analysis will typically reach a “dynamic” value in such situations. Shape analysis would then provide under-specified data, which would hinder further transformations that require concrete shape information, such as ASIC lowering.

Furthermore, some transformations proposed in the ML community are not well defined in the presence of control flow, such as the quantization transform described in Section 6.2.1.

```python
def loop_shapes(x, itr):
    # x is an input tensor of size [1, N]
    for _ in range(itr):
        x = torch.cat((x, x), dim=0)
    # Depending on the number of loop iterations, x may have an
    # arbitrary leading dimension i.e. x \in [\*dynamic, N]
    return x
```

Figure 4. A demonstration of dynamic shapes due to loop-carried dependencies

The fact that the IR does not contain control flow itself does not prevent transforms from working on sub-graphs of basic blocks within a larger model; We leave the details of how this composition works to the writer of the transform or the user applying the transform.

5.6 Functional Graphs but Stateful Modules

As described in Section 2.3, aliasing and mutability semantics in a language can necessitate complex analyses to prove that a program transformation is legal. torch.fx omits such analysis, instead defining mutating operations as undefined behavior with the option to raise errors when it is captured during tracing.

Avoiding mutability in the IR simplifies analysis and transformation of deep learning programs greatly. Most models do not suffer from this restriction since most mutation is localized to the parameters of the model.

torch.fx still preserves the hierarchical nn.Module structure from PyTorch and can represent module calls and attribute fetches from this structure. Modules like torch.nn.Conv2d are well understood by users, have well-documented arguments, and hide the stateful use of parameters within the module, so preserving these objects makes writing transformations easier. For instance, a torch.nn.BatchNorm module will actually contain mutable state, but that state is well understood by ML practitioners.

6 Case Studies and Evaluation

torch.fx has been used by PyTorch users both in the open-source ecosystem as well as a critical component of the deep learning stack at a major software company. We study the complexity of torch.fx’s IR and various use cases of torch.fx, including performance optimization, program analysis, and device and runtime export.

6.1 IR Complexity

One of the goals of torch.fx is to simplify the IR produced for ML models and make it easier for ML practitioners to understand. We can compare torch.fx IR to the IR produced
Aware Training requires access not only to parameter values but also to the activation values that flow through the program. The process for Quantization-Aware Training is analogous to phases (1) and (2) in the above but with “fake quantize” observers that snap floating point values to the corresponding values under quantized numerics.

6.2 Performance Optimization

PyTorch’s tensor language provides good performance in many cases, but architectural details of the underlying hardware create opportunities for further optimization. We investigate techniques by which \texttt{torch.fx} enables runtime performance improvements.

6.2.1 Quantization

Quantization (Jacob et al., 2017) is a technique used to increase the efficiency of neural network computation by reducing the size of Tensor data elements. Smaller data elements require less memory bandwidth, less storage, and can often be processed faster by modern processors. Neural network computation has relaxed sensitivity to numerical perturbations, so quantization is a canonical performance optimization.

Performing Post-Training Quantization or Quantization-Aware Training requires access not only to parameter values but also to the activation values that flow through the program (Krishnamoorthi, 2018). For instance, quantization-aware training needs to measure the distribution of floating point values in the output of a tensor addition operation to calculate a scale and bias value under quantized numerics. Such introspection is generally not available in PyTorch eager mode. However, \texttt{torch.fx} provides a lightweight way to capture such a program representation.

The Post-Training Quantization procedure entails the following stages:

1. A preparation phase, which instruments the program with “observer” objects that record statistical information about the floating-point values contained in Tensor values at various points in the program.
2. A calibration phase, where the user feeds batches of data through the network to populate the observers.
3. A conversion phase, where the collected statistics are used to down-cast weight values and convert operations in the model to quantized operations with embedded scale and zero-point information.

Quantization makes use of \texttt{torch.fx}’s graph and \texttt{GraphModule} representation to simultaneously modify the program code and weight values. The process for Post-Training Quantization applied on a server-class Intel Xeon Gold 6138 CPU @ 2.00GHz using FBGEMM (Kudzia et al., 2021) quantized operations. Figure 6 shows that \texttt{torch.fx}-enabled quantization confers up to a 3.3x runtime performance improvement compared to the floating point model, with low variance highlighting the predictable performance characteristics of ahead-of-time transformation. Correctness testing of quantization is not straightforward since it is a semantics-changing transform, but the applicability of numerics on this workflow has been validated on several model architectures via evaluation set testing. Numeric data for the experiment can be found in Appendix B. The preparation phase takes 44 ms, the calibration phase takes 590 ms, and the conversion phase takes 3.8 seconds. The majority of the time in the latter two phases can be attributed to tensor operations during model execution or value quantization, respectively.

Not only does \texttt{torch.fx}-based quantization provide the expected performance increases, but the tool’s development saw an order-of-magnitude productivity increase compared to an implementation on the TorchScript platform. By reducing the amount of complexity in the representation, exposing transformation APIs in Python, and embedding into the native PyTorch ecosystem, \texttt{torch.fx} provides a high-productivity environment for semantics-changing transforms like quantization.

6.2.2 Fusion Optimizations

Operator Fusion is a class of optimization that merges patterns of tensor operations together into a single compute
torch.fx: Practical Program Capture and Transformation for Deep Learning in Python

Figure 5. torch.fx traces through non-varying control flow and can embed constants as arguments in its Nodes. This substantially simplifies the IR for typical models. For a canonical ResNet50 model, torch.fx IR contains 445 operations compared to 2614 for torch.jit.script and 860 for torch.jit.trace.

Figure 6. Normalized inference runtime (lower is better) for torch.fx-based quantization.

One example of operator fusion is Convolution-BatchNorm fusion. During inference, a Convolution-BatchNorm operator sequence can be merged by applying the batch normalization weights to the convolution weights (Markuš, 2018).

We evaluate this transformation on a PyTorch ResNet50 model on an NVIDIA Tesla V100-SXM2 16GB with CUDA version 11.0 and an Intel Xeon Gold 6138 CPU @ 2.00GHz. Figure 7 shows approximately a 6% latency reduction for the GPU case, a 40% latency reduction on CPU with default intra-op parallelism, and a smaller 18% latency reduction with intra-op parallelism disabled (i.e. OMP_NUM_THREADS=1). Numerical correctness is confirmed via an epsilon equivalence comparison (rtol=1e-05, atol=1e-08) of the outputs of the fused and unfused implementations. Numeric results for this experiment can be found in Appendix C. The runtime of the transformation itself was 81 ms, the majority of which consists of the arithmetic operations to fuse the parameter tensors together.

torch.fx provides the necessary non-local program context and state modification facilities needed for this transformation with its ahead-of-time, graph-based nature (He, 2021). The whole transformation and test harness amount to fewer than 150 lines of Python, demonstrating the power of torch.fx’s APIs in enabling concise, fast-to-develop program transformations over PyTorch code.

Figure 7. Normalized inference runtime (lower is better) with torch.fx-based Convolution/Batch-Norm fusion.
6.2.3 Program Scheduling

Large PyTorch models sometimes contain blocking remote procedure calls to fetch values from parameter servers. For clarity these calls are written right before the parameters are used. However if a model contains several such calls, better utilization is achieved by overlapping these networks calls with other local work. With torch.fx, we provide a pass that replaces the blocking network calls with non-blocking ones and a separate wait call. We then hoist the non-blocking call as early as possible in the program. On large distributed training jobs, we have found this optimization can increase QPS by up to 9%.

6.3 Program Analysis

torch.fx has been applied in various ways for program analysis.

torch.fx has been used to implement a framework for simulation of deep learning inference at scale on various hardware devices at a major software company. torch.fx enables the estimation of FLOPs, memory bandwidth usage, and data value sizes of the workload, allowing for estimation of the program runtime and memory consumption. This system allows for rapid development of deep learning systems, enabling quick iteration in simulation rather than on real devices.

torch.fx has also been used for various forms of shape analysis. The canonical fx.passes.shape_prop package provides a naïve implementation of shape analysis by interpreting the graph and recording the observed shapes. Additional systems, including shape propagation via symbolic expressions and shape propagation via gradual typing semantics, are in development. torch.fx provides a representation on which such analyses can be done, opening opportunities for type system and inference innovations to be applied to PyTorch models.

Finally, torch.fx provides an fx.graph_drawer package, which gives the user the ability to visualize torch.fx graphs with Graphviz (Ellson et al., 2002). This provides a commonly-requested way of understanding a deep learning program via a visual representation of its DAG.

6.4 Device and Runtime Export/Compilation

PyTorch is primarily designed for modern GPUs, which provide a great deal of flexibility and dynamism and thus are very amenable to PyTorch’s eager mode execution model. However, GPUs can still benefit from ahead-of-time compilation of model code through toolkits like NVIDIA’s TensorRT (NVIDIA).

More specialized processors (such as the TPU (Jouppi et al., 2017)) promise higher performance, better power efficiency, and reduced cost via specialized functional units, specialized number formats, and new memory architectures. These processors often require static analyses and optimizations including operator scheduling, code generation, memory planning/scheduling, and architecture-aware quantization. Similarly to the optimizations in 6.2, such analyses typically require greater program context than the per-operator kernel launches provided by PyTorch during eager mode execution. torch.fx provides a pathway for such compiler stacks to integrate with PyTorch by providing a program representation extracted ahead-of-time. torch.fx is used at a major software company for ASIC lowering.

We evaluate lowering a PyTorch ResNet50 model and a LearningToPaint model (Huang et al., 2019) to NVIDIA TensorRT on an NVIDIA Tesla V100-SXM2 16GB GPU with CUDA version 11.0 using an experimental torch.fx-to-TensorRT lowering system. Figure 8 shows that TensorRT provides a predictable 3.7x runtime speed-up across 30 trials compared to baseline PyTorch for ResNet50 and a 1.54x speed-up for LearningToPaint. Numerical correctness is confirmed via an epsilon equivalence comparison (rtol=1e-05, atol=1e-08) of the outputs of the TensorRT and non-TensorRT implementations. Numerical data for this experiment is available in Appendix D.

In addition to providing the platform for runtime speed-up through TensorRT, torch.fx also provided high developer productivity for this component. The project was quickly developed using torch.fx’s Python APIs as well as TensorRT’s Python APIs, creating a translation layer between the two. The project was also able to quickly build components such as automatic splitting of the model based on TensorRT’s supported operators and automatically scheduling unsupported operations in non-optimized blocks. Finally, the ultimate user API is very easy to use, inspect, and debug, as it conforms to Python coding practices.

Figure 8. Normalized inference runtime (lower is better) with torch.fx-based TensorRT lowering
7 Conclusion

We presented torch.fx, a Python-only system for capturing and transforming PyTorch programs. We analyzed the factors that complicated related systems—including control flow, mutability, and data model—and show how torch.fx avoids complexity by focusing on common use cases and customizability. We investigated various use cases of torch.fx across optimization, analysis, and device lowering, and show how these results are enabled by torch.fx’s API design.

8 Acknowledgements

We would like to acknowledge all of the contributors to the core torch.fx framework, including Alban Desmaison, Alex Beloi, Alexander Soare, Allen (Congcong) Chen, Andrew Millsbaugh, Ansley Ussery, Aravind Kalaiyah, Bradley Davis, Brandon Lin, David Esiobu, Dmytro Dzhulgakov, Eli Uriegas, Erjia Guan, Garret Catron, Harut Movsisyan, Horace He, Hui Guo, James Reed, Jason Ansel, Jay Leverett, Jerry Cai, Jerry Zhang, Jordan Fix, Kefei Lu, Lu Fang, Malay Bag, Meghan Lele, Mehdi Mirzazadeh, Michael Benayoun, Michael Suo, Mike Ruberry, Mikhail Zolotukhin, Natalia Gimelshein, Nikita Shulga, Oleg Khabinov, Onyee, Patrick Hu, Patrick Spencer, Peter Bell, Philip Meier, Richard Zou, Sam Estep, Shirong Wu, Shiyan Deng, Thomas Wang, Vasily Kuznetsov, Yinghai Lu, Zachary DeVito, and Zeina Migeed.

We would like to acknowledge the contributors to the FX Graph Mode Quantization project used in evaluations, including Adnios, Alban Desmaison, Angela Yi, Bradley Davis, Charles David Hernández, Emily Shen, Erjia Guan, Horace He, James Reed, Jerry Zhang, Raghuraman Krishnamoorthi, Mike Ruberry, Mikhail Zolotukhin, Philip Meier, Rong Rong (AI Infra), Sam Estep, Supriya Rao, Vasily Kuznetsov, Xiang Gao, Zachary DeVito, and Zafar Takhirov.

We would like to acknowledge the contributors to the fx2trt project used in evaluations, including Alex Beloi, Aravind Kalaiyah, Bangsheng Tang, Eli Uriegas, Emad El-Haraty, Ivan Kobzarev, Jack Montgomery, James Reed, Jerry Zhang, Jordan Fix, Kefei Lu, Linbin Yu, Matr Subkhankulov, Mike Ruberry, Mor Tzur, Nikita Shulga, Philip Meier, Protonu Busu, Rui Zhu, Samuel Salas, Shirong Wu, Shiyan Deng, Yinghai Lu, and Zhengxu Chen.

Finally, we’d like to acknowledge all the discussions and feedback from many users inside and outside Facebook.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 265–283, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

Abbasi, H., Yang, E. Z., and Gommero, R. Improving subclassing Tensor by propagating subclass instances, Aug 2020. URL https://github.com/pytorch/rfcs/blob/master/RFC-0001-torch-function-for-methods.md.

Agrawal, A., Modi, A. N., Passos, A., Lavoie, A., Agarwal, A., Shankar, A., Ganichev, I., Levenberg, J., Hong, M., Monga, R., and Cai, S. TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning. CoRR, abs/1903.01855, 2019. URL http://arxiv.org/abs/1903.01855.

Al-Rfou, R., Alain, G., Almahairi, A., Angermüller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., Bengio, Y., Bergeron, A., Bergrstra, J., Bisson, V., Bouchard, N., Boulanger-Lewandowski, N., Bouthillier, X., de Brébisson, A., Breuleux, O., Carrier, P. L., Cho, K., Chorowski, J., Christiano, P. F., Cooijmans, T., Côté, M., Côté, M., Courville, A. C., Dauphin, Y. N., Delalleau, O., Demouth, J., Desjardins, G., Dieleman, S., Dinh, L., Duchﬀe, M., Dumoulin, V., Kahou, S. E., Erhan, D., Fan, Z., Firat, O., Germain, M., Glorot, X., Goodfellow, I. J., Graham, M., Gülçehre, Ç., Hamel, P., Harlouchet, I., Heng, J., Hidasi, B., Honari, S., Jain, A., Jean, S., Jia, K., Kobrov, M., Kulkarni, V., Lamb, A., Lamblin, P., Larsen, E., Laurent, C., Lee, S., Lefrançois, S., Lemieux, S., Léonard, N., Lin, Z., Livezey, J. A., Lorenz, C., Lowin, J., Ma, Q., Manzagol, P., Mastroianni, O., McGibbon, R., Memisevic, R., van Merriënboer, B., Michalski, V., Mirza, M., Orlandi, A., Pal, C. J., Pascanu, R., Pezeshki, M., Raffel, C., Renshaw, D., Rocklin, M., Romero, A., Roth, M., Sadowski, P., Salvatier, J., Savard, F., Schlüter, J., Schulman, J., Schwartz, G., Serban, I. V., Serdyuk, D., Shabanian, S., Simon, É., Spieckermann, S., Subramanyam, S. R., Synowiecki, J., Tanguay, J., van Tulder, G., Turian, J. P., Urban, S., Vincent, P., Visin, F., de Vries, H., Warde-Farley, D., Webb, D. J., Willson, M., Xu, K., Xue, L., Yao, L., Zhang, S., and Zhang, Y. Theano: A Python framework for fast computation of mathematical expressions. CoRR, abs/1605.02688, 2016. URL http://arxiv.org/abs/1605.02688.

Andersen, L. O. Program analysis and specialization for the C programming language. PhD thesis, Citeseer, 1994.
Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078.

DeVito, Z. et al. TorchScript, Sep 2018. URL https://pytorch.org/docs/1.9.0/jit.html.

Ellson, J., Gansner, E., Koutsofios, L., North, S. C., and Woodhull, G. Graphviz— Open Source Graph Drawing Tools. In Mutzel, P., Jünger, M., and Leipert, S. (eds.), *Graph Drawing*, pp. 483–484, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45848-7.

Elman, J. L. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

Fischer, K. and Saba, E. Automatic Full Compilation of Julia Programs and ML Models to Cloud TPUs. *CoRR*, abs/1810.09868, 2018. URL http://arxiv.org/abs/1810.09868.

Frostig, R., Johnson, M. J., and Leacy, C. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.

He, H. (Beta) building a convolution/batch norm fuser in fx, Mar 2021. URL https://pytorch.org/tutorials/intermediate/fx_conv_bn_fuser.html.

He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Heek, J., Levskaya, A., Oliver, A., Ritter, M., Ronde pierre, B., Steiner, A., and van Zee, M. Flax: A neural network library and ecosystem for JAX, 2020. URL http://github.com/google/flax.

Hennigan, T., Cai, T., Norman, T., and Babuschkin, I. Haiku: Sonnet for JAX, 2020. URL http://github.com/deepmind/dm-haiku.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Huang, Z., Heng, W., and Zhou, S. Learning to paint with model-based deep reinforcement learning. *CoRR*, abs/1903.04411, 2019. URL http://arxiv.org/abs/1903.04411.

Innes, M. Don’t Unroll Adjoint: Differentiating SSA-Form Programs. *CoRR*, abs/1810.07951, 2018. URL http://arxiv.org/abs/1810.07951.

Innes, M. et al. On machine learning and programming languages, Dec 2017. URL https://julialang.org/blog/2017/12/ml-pl/.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A. G., Adam, H., and Kalenichenko, D. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *CoRR*, abs/1712.05877, 2017. URL http://arxiv.org/abs/1712.05877.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R. B., Guadarrama, S., and Darrell, T. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR*, abs/1408.5093, 2014. URL http://arxiv.org/abs/1408.5093.

Kildall, G. A. *Global expression optimization during compilation*. University of Washington, 1972.

Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342, 2018. URL http://arxiv.org/abs/1806.08342.

Kuchaiev, O. and Ginsburg, B. Training deep autoencoders for collaborative filtering, 2017.

Kudlur, M. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

Khudia, D. S., Huang, J., Basu, P., Deng, S., Liu, H., Park, J., and Smelyanskiy, M. FBGEMM: Enabling High-Performance Low-Precision Deep Learning Inference. *CoRR*, abs/2101.05615, 2021. URL https://arxiv.org/abs/2101.05615.

Kildall, G. A. *Global expression optimization during compilation*. University of Washington, 1972.

Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342, 2018. URL http://arxiv.org/abs/1806.08342.

Kuchaiev, O. and Ginsburg, B. Training deep autoencoders for collaborative filtering, 2017.

Lattner, C., Pienaar, J. A., Amini, M., Bondhugula, U., Riddle, K., Cohen, A., Shpeisman, T., Davis, A., Vasilache, N., and Zinenko, O. MLIR: A Compiler Infrastructure for the End of Moore’s Law. *CoRR*, abs/2002.11054, 2020. URL https://arxiv.org/abs/2002.11054.

Maclaurin, D., Duvenaud, D., and Adams, R. P. Auto grad: Effortless Gradients in Numpy. In *ICML 2015 AutoML Workshop*, 2015. URL /bib/maclaurin/
Markuš, N. Fusing batch normalization and convolution in runtime, May 2018. URL https://nenadmarkus.com/p/fusing-batchnorm-and-conv/.

Moldovan, D., Decker, J. M., Wang, F., Johnson, A. A., Lee, B. K., Nado, Z., Sculley, D., Rompf, T., and Wiltschko, A. B. AutoGraph: Imperative-style Coding with Graph-based Performance. CoRR, abs/1810.08061, 2018. URL http://arxiv.org/abs/1810.08061.

Naumov, M., Mudigere, D., Shi, H. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M. Deep Learning Recommendation Model for Personalization and Recommendation Systems. CoRR, abs/1906.00991, 2019. URL http://arxiv.org/abs/1906.00991.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Amar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., et al. Dynet: The dynamic neural network toolkit. arXiv preprint arXiv:1701.03980, 2017.

NVIDIA. Abstract. URL https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. CoRR, abs/1912.01703, 2019. URL http://arxiv.org/abs/1912.01703.

Rompf, T. and Odersky, M. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In Proceedings of the ninth international conference on Generative programming and component engineering, pp. 127–136, 2010.

Saeta, B., Shabalin, D., Rasi, M., Larson, B., Wu, X., Schuh, P., Casbon, M., Zheng, D., Abdurrasool, S., Efremov, A., Abrahams, D., Lattner, C., and Wei, R. Swift for TensorFlow: A portable, flexible platform for deep learning. CoRR, abs/2102.13243, 2021. URL https://arxiv.org/abs/2102.13243.

Suhan, A., Libenzi, D., Zhang, A., Schuh, P., Saeta, B., Sohn, J. Y., and Shabalin, D. LazyTensor: combining eager execution with domain-specific compilers. arXiv preprint arXiv:2102.13267, 2021.

The XLA Team. XLA - tensorflow, compiled, Mar 2017. URL https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.

Tokui, S., Okuta, R., Akiba, T., Niitani, Y., Ogawa, T., Saito, S., Suzuki, S., Uenishi, K., Vogel, B., and Vincent, H. Y. Chainer: A Deep Learning Framework for Accelerating the Research Cycle. CoRR, abs/1908.00213, 2019. URL http://arxiv.org/abs/1908.00213.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention Is All You Need. CoRR, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Xiao, F. et al. Protocolbuffers/Protobuf: Protocol buffers - google’s data interchange format, 2008. URL https://github.com/protocolbuffers/protobuf.

A TORCH.FX NODE SEMANTICS

A.1 Opcode Meanings

| Opcode       | Meaning                                      |
|--------------|----------------------------------------------|
| placeholder  | Function Input                               |
| call_method  | Call method on args[0]                      |
| call_module  | Call module specified by target              |
| call_function| Call function specified by target            |
| get_attr     | Retrieve attribute specified by target       |
| output       | Return statement; return args[0]             |

A.2 args/kwargs Behavior

| Opcode       | args/kwargs Behavior                          |
|--------------|-----------------------------------------------|
| placeholder  | Empty or args[0] = default value              |
| call_method  | Python calling convention; args[0] is self    |
| call_module  | Python calling convention; target is self    |
| call_function| Python calling convention; target is self    |
| get_attr     | Empty                                         |
| output       | args[0] is the return value                   |
## B Quantization Evaluation Numeric Data

| Batch Size | Runtime Unquantized | stdev Unquantized | Runtime Quantized | stdev Quantized |
|------------|---------------------|-------------------|-------------------|-----------------|
| 1          | 0.0777              | 0.00079           | 0.0222            | 0.0008          |
| 16         | 0.1980              | 0.0104            | 0.0639            | 0.0057          |
| 64         | 0.3995              | 0.0204            | 0.2585            | 0.0129          |
| 128        | 0.6717              | 0.0228            | 0.5369            | 0.0413          |
| 256        | 1.2307              | 0.0874            | 1.1137            | 0.0686          |

## C Fusion Evaluation Numeric Data

| Device | Fusion | Threads | Average run-time (sec) | stdev run-time |
|--------|--------|---------|------------------------|----------------|
| GPU    | Unfused| N/A     | 0.1887                 | 0.00048        |
| GPU    | Fused  | N/A     | 0.1777                 | 0.00049        |
| CPU    | Unfused| Threaded| 0.2996                 | 0.02835        |
| CPU    | Fused  | Threaded| 0.2129                 | 0.03491        |
| CPU    | Unfused| Unthreaded| 2.0231                 | 0.23050        |
| CPU    | Fused  | Unthreaded| 1.7166                 | 0.25091        |

## D TensorRT Evaluation Numeric Data

| Configuration   | Avg Run-time (sec) | Stdev Run-time |
|-----------------|--------------------|----------------|
| PyTorch RN50    | 0.2443             | 0.00119        |
| torch.fx TensorRT RN50 | 0.0662         | 0.00022        |
| PyTorch LearningToPaint | 0.0068         | 0.0003         |
| torch.fx TensorRT LearningToPaint | 0.0044         | 0.0001         |