Abstract

PyTorch is one of the most commonly used deep learning libraries which performs strong acceleration on graphic processor units. The work revolves around enabling type promotion support for a class of element wise functions in PyTorch, also known as unary functions. Promoting data types of input tensors within PyTorch's implementation of it's tensors prevents extra kernel launches and thus improves performance on graphic processor units (GPUs). The decision of which function should support type promotion and what rules should be followed and implemented, is reviewed by comparing type promotion strategies of common scientific computation libraries like NumPy and JAX. PyTorch, having a high velocity codebase, goes through significant changes every day and a lot of it's source code is not documented for a reader to understand in the first read. The work helps describe relevant parts of PyTorch internals as well. Allowing type promotion for relevant unary functions, allows to implicitly promote the integral and boolean input tensors to floating point tensors and thus prevents any run time error. In the open source community, most of the improvements are made depending on the user's requirements. Type Promotion for Unary Functions in PyTorch has been highlighted many times and thus, was prioritized after the work was initiated. The work also helped initiate the cleaning up of PyTorch's code base to help support better changes as per agile development guidelines. Since PyTorch is used on many platforms, and not just one, it's important for any change in the code base to go through several tests. The work also explains the test cases written in PyTorch for type promotion, and how different components of PyTorch are looked after before any change is accepted and merged into the master branch of the library.

Keywords: Deep Learning, PyTorch, Graphic Processor Unit (GPU), Type Promotion, Scientific Computation

Contents

List of Abbreviations

ONNX	:	Open Neural Network Exchange
JIT	:	Just In Time

List of Figures

List of Tables

CHAPTER 1

Introduction

Preface

This chapter covers the introduction of the thesis, motivation behind the work, a survey of the reported issues related to the work done and organization of the thesis. The impact of the work in different fields, industries and research are also discussed. This chapter will give insights on PyTorch internals, how NVIDIA comes into the frame and how the work could effectively make things easier and better for the stakeholders of PyTorch. This chapter also sets the context on type promotion and unary universal functions.

1.1 Introduction

The work discussed in this thesis largely revolves around PyTorch[1], a library owned and maintained by Facebook focused to provide high performance on Graphic Processor Units (also known as GPUs) provided by NVIDIA for computations with tensors (a type of multi-dimensional arrays), and a sophisticated implementation of deep neural networks with an automatic differentiation (also known as autograd) system. Since the first alpha release[2] of PyTorch (1st September 2016), the demand of Deep Learning and auto differentiation (also known as autograd) modules has grown rapidly, mostly dealt to solve Computer Vision, Speech Processing and Artificial Intelligence focused problems.

1.2 Overview of PyTorch

While PyTorch is mainly maintained by the developers and researchers at Facebook AI Research (FAIR)[3], it highly relies on the community for contributions and companies like NVIDIA, AMD and Intel to help them provide better performance and make PyTorch ready to use on their respective hardwares. The work described in the thesis has been done during the internship at NVIDIA (Santa Clara, California, Headquarters) in the PyTorch Development Team and the author along with the PyTorch Development Team have attempted to help initiate the idea of type promotion for Unary Universal Functions and implement that for the available platforms which will be discussed later, including NVIDIA's Deep Learning High Performance Computing Platform (CUDA)[4].

PyTorch has been inherently motivated by NumPy[5], one of the most popular scientific computational libraries available in Python. The implementation and organization of PyTorch,

it's features have been highly motivated by NumPy, ranging from how NumPy implements tensors, to its type promotion strategies, almost all the related discussions to PyTorch are well thought of before bringing them into practice. This is not just to help users port their code from NumPy to PyTorch but also make the code readable and reproducible. This also motivates the importance of keeping standards and not deviating away from the standard maintained by already existing, popular, stable and commonly used libraries like NumPy.

Researchers, and industries heavily rely on PyTorch, it's behavior and performance on various devices like Raspberry Pi and embedded devices with different processors. Every feature request from a user has to be discussed both internally and externally with the community, to make sure it doesn't produce any adverse effects on the existing customers and stakeholders. While many use PyTorch in Python, the source code is written in C++ and CUDA for better performance, and speedup. Most of the libraries follow the same standard of writing source code in low level programming languages like C or C++, and using Python on the top level for better readability and easier usage. Over the last year, Facebook has started focusing on bringing the C++ API of PyTorch (also called Libtorch)[6] closer to what Python API looks like.

1.3 Motivation

While the work broadly revolves around Unary Universal Functions of PyTorch, the connection goes down to the low level implementation of PyTorch tensors (also called TensorIterator API) which is still one of the most complicated and undocumented part of PyTorch. This work attempts to make it easier for the readers to understand how TensorIterator promotes types of unary universal functions and how this helps improve performance. Type Promotion has not just been an integral part of any library that deals with deep learning, auto differentiation and computing, but most discussed as well. One of the best examples of this would be a user passing an integer to a cosine function in PyTorch and getting an error that it can only operate on floating point tensors. The error makes sense because the computation has to be done in floating point format, and integer or long formats are not preferred. But, for a library to be stable and user-centric, it has to be able to allow the users to give integer inputs and expect floating point outputs on such functions.

The motivation of this work was to be able to promote data types of inputs without an extra kernel launch, which is discussed in the later part of this thesis. This prevents significant overhead for large scale problems, which is significant in case of parameter updates in deep learning like applications. While, initially, the target was to have a keyword only argument for the unary universal functions in PyTorch for users to explicitly promote the input data type to given data type, but the project turned out to be a motivation to change the existing type promotion strategy for these functions in PyTorch. The challenge has been not just to be able to promote data types implicitly, but also to help move type promotion discussion forward. It

has, since then, been a hot topic discussed in the community.

1.4 Organization of the Thesis

The organization of this project is done in the following manner:

Chapter 1 sets the context and gives a brief introduction to PyTorch, how NVIDIA comes into the picture and a brief description about the problem statement. It is also aimed to outline the organization of this thesis and methodology followed by the author.

Chapter 2 describes the relevant parts of PyTorch internals (source code) and discusses the approach on understanding the problem statement. The chapter attempts to guide the reader from the very basics of PyTorch internals to the TensorIterator API.

Chapter 3 describes the existing PyTorch TensorIterator API and the type promotion strategy used in PyTorch before the beginning of this project. It attempts to explain one of the most complex parts of PyTorch in easy to understand and pictographic fashion.

Chapter 4 talks about the comparison of strategies opted by JAX[7, 8] and NumPy for type promotion of unary functions. The chapter also discusses the implementation of type promotion for some of the unary functions in PyTorch. It also discusses the tests, changes done in JIT and ONNX sections of PyTorch. The chapter is highly specific in terms of code, and it's suggested to the reader to go through the previous chapters before this.

Chapter 5 gives a brief overview of the contributions made by the author to the PyTorch library. It links to the pull requests made in an attempt to make PyTorch better for the relevant use cases discussed in the previous chapters.

CHAPTER 2

Understanding PyTorch Internals

Preface

This chapter focuses on the understanding of PyTorch internals, and how it is organized. In order to contribute to any open source library like PyTorch, it's highly suggested to first be comfortable with the conventions followed and the design patterns used. PyTorch follows it's own dispatch routines, as the source code is written in C++ and it's worth noting how the Python API dispatches the call to the implementation in C++. This is discussed in the chapter using available debugging tools.

2.1 Introduction

The aim of this chapter is to describe the internals of PyTorch which are relevant to enabling type promotion to unary functions in PyTorch. Just like any other sophisticated library written in C++, PyTorch also contains a lot of unexplained parts like dispatch macros and it's thus important to discuss the design patterns followed. While most of the stakeholders of PyTorch use Python, the source code is written in C++ and it's thus important to understand how the dispatch routine works in PyTorch from python call to the source code in C++.

There are many debuggers available, *gdb*[9] being one of the most popular ones to debug C++ libraries. In order to reproduce results, the reader is encouraged to build PyTorch from source based on the guidelines given here[10] using debug symbols (*DEBUG=1*). *gdb* debugger gives an option to trace back the code from the actual Python call to the function's definition written in C++. This will help observing which functions are called during the dispatch.

2.2 Back Tracing to Understand Gradient Storing in PyTorch

Understanding how PyTorch dispatch works, and how TensorIterator comes into play using gdb backtracking:

```
sample_tensor = torch.tensor([1, 2, 3], dtype=torch.float32, requires_grad=True)
torch.log_softmax(sample_tensor, dim=0)
```

Figure 1.1: Code snippet used to back trace the *log_softmax* call to understand how gradients are stored in PyTorch internally. This gives a brief introduction to the dispatch routine in PyTorch, and how it operates with an argument like *requires_grad*.

Table 1.1 shows the back tracing output from *gdb* debugger.

S.No	Function Description
1	*torch::autograd::THPVariable_log_softmax (self_=<Tensor at remote 0x7fff6b993510>, args=(0,), kwargs=0x0)* Location: torch/csrc/autograd/generated/python_variable_methods.cpp:1459
2	*torch::autograd::dispatch_log_softmax (self=..., dim=0, dtype=...)* Location: torch/csrc/autograd/generated/python_variable_methods_dispatch.h:1361
3	*at::Tensor::log_softmax (this=0x7fff6b993520, dim=0, dtype=...)* Location: build/aten/src/ATen/core/TensorMethods.h:1582
4	*c10::OperatorHandle::callUnboxed<at::Tensor, at::Tensor const&, long, c10::optional<c10::ScalarType> > (this=0x7fffc5748ef0 <at::Tensor::log_softmax(long, c10::optional<c10::ScalarType>) const::op>, args#0=..., args#1=0, args#2=...)* Location: aten/src/ATen/core/dispatch/Dispatcher.h:173
5	*c10::Dispatcher::callUnboxed<at::Tensor, at::Tensor const&, long, c10::optional<c10::ScalarType> > (this=0x7fffc21eec80 <c10::Dispatcher::singleton()::_singleton>, op=..., args#0=..., args#1=0, args#2=...)*

6

	Location: aten/src/ATen/core/dispatch/Dispatcher.h:210
6	*c10::Dispatcher::callUnboxedWithDispatchKey<at::Tensor, at::Tensor const&, long, c10::optional<c10::ScalarType> > (* *this=0x7fffc21eec80 <c10::Dispatcher::singleton()::_singleton>, op=...,* *dispatchKey=..., args#0=..., args#1=0, args#2=...)* Location: aten/src/ATen/core/dispatch/Dispatcher.h:202
7	*c10::KernelFunction::callUnboxed<at::Tensor, at::Tensor const&, long, c10::optional<c10::ScalarType> > (this=0x13603d8,* *opHandle=..., args#0=..., args#1=0, args#2=...)* Location: aten/src/ATen/core/boxing/KernelFunction_impl.h:62
8	*c10::detail::wrap_kernel_functor_unboxed_<c10::detail::WrapRuntimeKernelFunctor_<at::Tensor (*)(at::Tensor const&, long, c10::optional<c10::ScalarType>), at::Tensor, c10::guts::typelist::typelist<at::Tensor const&, long, c10::optional<c10::ScalarType> > >, at::Tensor (at::Tensor const&, long, c10::optional<c10::ScalarType>)>::call(c10::OperatorKernel*, at::Tensor const&, long, c10::optional<c10::ScalarType>) (functor=0x1949580, args#0=..., args#1=0, args#2=...)* Location: aten/src/ATen/core/boxing/kernel_functor.h:262
9	*c10::detail::WrapRuntimeKernelFunctor_<at::Tensor (*)(at::Tensor const&, long, c10::optional<c10::ScalarType>), at::Tensor, c10::guts::typelist::typelist<at::Tensor const&, long, c10::optional<c10::ScalarType> > >::operator() (this=0x1949580, args#0=..., args#1=0, args#2=...)* Location: aten/src/ATen/core/boxing/kernel_lambda.h:23
10	*torch::autograd::VariableType::(anonymous namespace)::log_softmax (self=..., dim=0, dtype=...)* Location: torch/csrc/autograd/generated/VariableType_2.cpp:6043
11	*at::TypeDefault::log_softmax (self=..., dim=0, dtype=...)* Location: build/aten/src/ATen/TypeDefault.cpp:1979

12	*at::native::log_softmax (input_=..., dim_=0, dtype=...)* Location: aten/src/ATen/native/SoftMax.cpp:275
13	*at::native::<lambda()>::operator()(void) const (* *__closure=0x7fffffffd2b0)* Location: aten/src/ATen/native/SoftMax.cpp:273
14	*at::_log_softmax (self=..., dim=0, half_to_float=false)* Location: build/aten/src/ATen/Functions.h:5230
15	*c10::OperatorHandle::callUnboxed<at::Tensor, at::Tensor const&, long, bool> (this=0x7fffc21f2940 <at::_log_softmax(at::Tensor const&, long, bool)::op>, args#0=..., args#1=0, args#2=false)* Location: aten/src/ATen/core/dispatch/Dispatcher.h:173
16	*c10::Dispatcher::callUnboxed<at::Tensor, at::Tensor const&, long, bool> (this=0x7fffc21eec80 <c10::Dispatcher::singleton()::_singleton>, op=..., args#0=..., args#1=0, args#2=false)* Location:aten/src/ATen/core/dispatch/Dispatcher.h:210
17	*c10::Dispatcher::callUnboxedWithDispatchKey<at::Tensor, at::Tensor const&, long, bool> (* *this=0x7fffc21eec80 <c10::Dispatcher::singleton()::_singleton>, op=...,* *dispatchKey=..., args#0=..., args#1=0, args#2=false)* Location: aten/src/ATen/core/dispatch/Dispatcher.h:202
18	*c10::KernelFunction::callUnboxed<at::Tensor, at::Tensor const&, long, bool? (this=0x104ef18, opHandle=..., args#0=..., args#1=0, args#2=false)* Location: aten/src/ATen/core/boxing/KernelFunction_impl.h:62
19	*c10::detail::wrap_kernel_functor_unboxed_<c10::detail::WrapRuntimeKernelFunctor_<at::Tensor (*)(at::Tensor const&, long, bool), at::Tensor, c10::guts::typelist::typelist<at::Tensor const&, long, bool> >, at::Tensor (at::Tensor const&, long, bool)>::call(c10::OperatorKernel*, at::Tensor const&, long, bool) (functor=0x1706440, args#0=..., args#1=0, args#2=false)*

	Location: aten/src/ATen/core/boxing/kernel_functor.h:262
20	*c10::detail::WrapRuntimeKernelFunctor_<at::Tensor (*)(at::Tensor const&, long, bool), at::Tensor, c10::guts::typelist::typelist<at::Tensor const&, long, bool> >::operator() (this=0x1706440, args#0=..., args#1=0, args#2=false)* Location: aten/src/ATen/core/boxing/kernel_lambda.h:23
21	*torch::autograd::VariableType::(anonymous namespace)::_log_softmax (self=..., dim=0, half_to_float=false)* Location: torch/csrc/autograd/generated/VariableType_2.cpp:519
22	*torch::autograd::VariableType::(anonymous namespace)::<lambda()>::operator()(void) const (__closure=0x7fffffffcd90)* Location: torch/csrc/autograd/generated/VariableType_2.cpp:518
23	*at::_log_softmax (self=..., dim=0, half_to_float=false)* Location: build/aten/src/ATen/Functions.h:5230
24	*c10::OperatorHandle::callUnboxed<at::Tensor, at::Tensor const&, long, bool> (this=0x7fffc220ba08 <_ZZN2atL12_log_softmaxERKNS_6TensorElbE2op>, args#0=..., args#1=0, args#2=false)* Location: aten/src/ATen/core/dispatch/Dispatcher.h:173
25	*c10::Dispatcher::callUnboxed<at::Tensor, at::Tensor const&, long, bool> (this=0x7fffc21eec80 <c10::Dispatcher::singleton()::_singleton>, op=..., args#0=..., args#1=0, args#2=false)* Location:aten/src/ATen/core/dispatch/Dispatcher.h:210
26	*c10::Dispatcher::callUnboxedWithDispatchKey<at::Tensor, at::Tensor const&, long, bool> (this=0x7fffc21eec80 <c10::Dispatcher::singleton()::_singleton>, op=..., dispatchKey=..., args#0=..., args#1=0, args#2=false)* Location: aten/src/ATen/core/dispatch/Dispatcher.h:202

27	*c10::KernelFunction::callUnboxed<at::Tensor, at::Tensor const&, long, bool> (this=0x104ea98, opHandle=..., args#0=..., args#1=0, args#2=false)* Location: aten/src/ATen/core/boxing/KernelFunction_impl.h:62
28	*c10::detail::wrap_kernel_functor_unboxed_<c10::detail::WrapRuntimeKernelFunctor_<at::Tensor (*)(at::Tensor const&, long, bool), at::Tensor, c10::guts::typelist::typelist<at::Tensor const&, long, bool> >, at::Tensor (at::Tensor const&, long, bool)>::call(c10::OperatorKernel*, at::Tensor const&, long, bool) (functor=0x1b93e50, args#0=..., args#1=0, args#2=false)* Location: aten/src/ATen/core/boxing/kernel_functor.h:262
30	*c10::detail::WrapRuntimeKernelFunctor_<at::Tensor (*)(at::Tensor const&, long, bool), at::Tensor, c10::guts::typelist::typelist<at::Tensor const&, long, bool> >::operator() (this=0x1b93e50, args#0=..., args#1=0, args#2=false)* Location: aten/src/ATen/core/boxing/kernel_lambda.h:23
31	*at::CUDAType::(anonymous namespace)::_log_softmax (self=..., dim=0, half_to_float=false)* Location: build/aten/src/ATen/CUDAType.cpp:814
32	*at::native::log_softmax_cuda(at::Tensor const&, long, bool)* Location: build/aten/src/ATen/CUDAType.cpp:814

Table 1.1: Output of back tracing *log_softmax* call in Python API of PyTorch on a tensor with *requires_grad* argument set as *True*. Each cell is titled with the function declaration (which is called) along with the function's location for reference, and it's inference (if any).

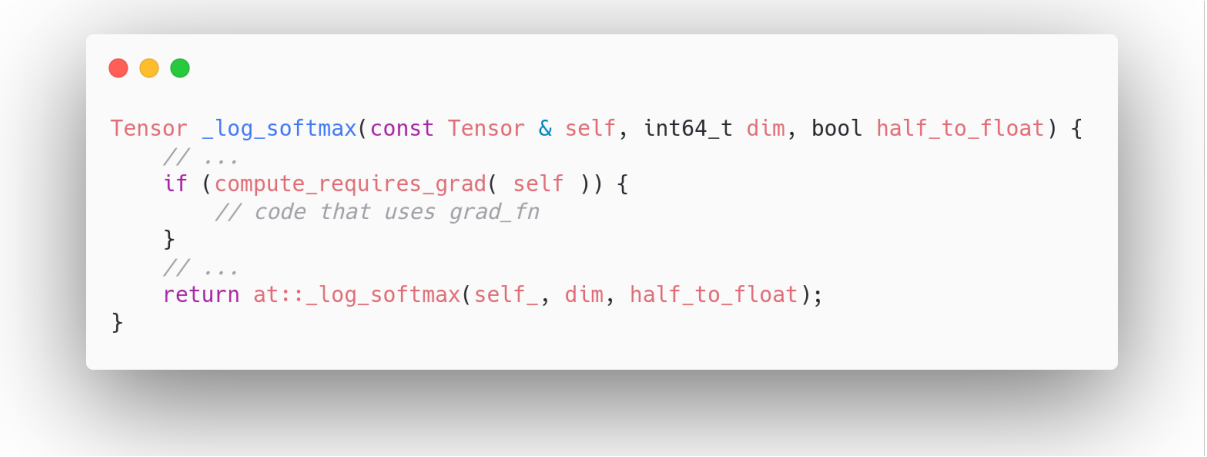
The conclusions made from the Table 1.1 are as follows:

1. The Python function *log_softmax* is binded to the C++ Function *THPVariable_log_softmax* here: *torch/csrc/autograd/generated/python_variable_methods.cpp* (LINE 1184).

```
{"log_softmax", (PyCFunction)(void(*)(void))THPVariable_log_softmax, METH_VARARGS | METH_KEYWORDS |
METH_STATIC, NULL}
```

Figure 1.2: Code snippet of how log_softmax function in Python is dispatched to C++ dispatch routine

2. If the PyTorch is not built in the debug mode, it skips putting the function calls in the stack for backtrace, but if it's built in the debug mode - it creates memory in the stack for backtrace, and also outputs the debug symbols (and source code paths etc.).
3. It's important to understand where the autograd history recording takes place. In the function *_log_softmax()* located at *torch/csrc/autograd/generated/VariableTypeEverything.cpp*, it's definition's template is:



```
Tensor _log_softmax(const Tensor & self, int64_t dim, bool half_to_float) {
    // ...
    if (compute_requires_grad( self )) {
        // code that uses grad_fn
    }
    // ...
    return at::_log_softmax(self_, dim, half_to_float);
}
```

Figure 1.3: The code snippet showing definition of *_log_softmax* which is automatically generated. The code which is not relevant here has been commented out. The intention is to note down the step where autograd history recording takes place.

4. The condition to check if *_compute_requires_grad(self)* is *True* or not in the Figure 1.3 above is explained in Figure 1.4.

```
if (compute_requires_grad( self )) {
    grad_fn = std::shared_ptr<LogSoftMaxBackward>(new LogSoftmaxBackward(), deleteNode);
    grad_fn->set_next_edges(collect_next_edges( self ));
    grad_fn->self_ = SavedVariable(self, false);
    grad_fn->dim = dim;
}
// ...
auto temp = ([&]() {
    at::AutoNonVariableTypeMode non_var_type_mode(true);
    return at::_log_softmax(self_, dim, half_to_float);
})();

auto result = std::move(tmp);
torch::jit::Node* node = nullptr;
if (grad_fn) {
    set_history(flatten_tensor_args( result ), grad_fn);
}
if (tracer_state) {
    // …
}
if (grad_fn) {
    grad_fn->result_ = SavedVariable(result, true);
}
```

Figure 1.4: The part of code showing how autograd history is recorded in PyTorch. The checks of *if (grad_fn)* are the places where gradient history is recorded.

5. Initially, (if the argument, *requires_grad* is set *True*) *grad_fn* node is created of type *LogSoftMaxBackward()*.
6. Then the properties of the nodes are set using *grad_fn->set_next_edges, grad_fn->self_ and grad_fn->dim*.
7. The result of *log_softmax* is stored in a variable *tmp* and then moved to *result*.
8. If *grad_fn* is not *NULL*, then the history is recorded and the result is saved to *grad_fn->result_*.

The aim of the above exercise was to understand PyTorch internals and be able to back trace to the place where autograd history is recorded. Since the end goal is to understand the type promotion strategy of PyTorch and enable it for Unary Universal Functions, it is worth looking at the back tracing output with a *dtype* flag.

2.3 Understanding PyTorch Dispatch Macros

To repeat, the intention is to be more comfortable with PyTorch internals as most of the source code is not documented and most of the time, one has to scroll through the pull requests and the discussions to understand why a certain feature was introduced. One of the sophisticated design conventions in any large scale library like PyTorch is the use of macros..

The usage of macros is comparatively complex, to what seen in day to day use. The Dispatch Macros explained below are used to expand a given function definition (through lambda functions) to specified data types. Think of implementing cosine function in PyTorch, and adding support for complex types, floating point data types and integer types only, that's where these macros come into the frame.

```
void sin_kernel_cuda(TensorIterator& iter) {
  AT_DISPATCH_FLOATING_AND_COMPLEX_TYPES_AND1(ScalarType::Half, iter.dtype(), "sin_cuda", [&]() {
    using thrust_t = typename ztype_cuda<scalar_t>::thrust_t;
    gpu_kernel(iter, []GPU_LAMBDA(thrust_t a) -> thrust_t {
      return sin_wrapper(a);
    });
  });
}
```

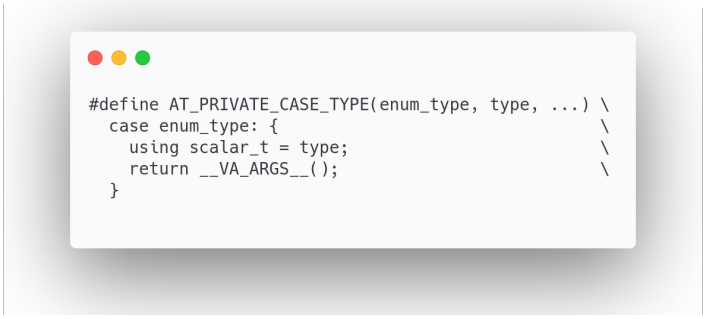
Figure 1.5. The figure demonstrates the usage of dispatch macros in implementation of a function's kernel. *AT_DISPATCH_FLOATING_AND_COMPLEX_TYPES_AND1* is demonstrated in the implementation of sine function (for CUDA). The macro is used to allow all floating point data types, complex types and floating point 16 (used as *ScalarType::Half*) for the given function.

Taking a look at the definition of the used macro in Figure 1.5, makes it easier to understand how these macros prevent repetition of code throughout PyTorch source code. These macros are heavily used throughout the PyTorch source code and it's thus highly recommended to understand how they function. To help better understand the usage of these macros, their definitions and a sample usage code is discussed.

These macros are of form *AT_DISPATCH_ALL_TYPES_AND_COMPLEX_ANDN (1 <= N <= 3)*.

```
#define AT_DISPATCH_FLOATING_AND_COMPLEX_TYPES_AND1(SCALARTYPE, TYPE, NAME, ...)
        \
  [&] {
        \
    const auto& the_type = TYPE;
         \
    /* don't use TYPE again in case it is an expensive or side-effect op */
          \
    at::ScalarType _st = ::detail::scalar_type(the_type);
          \
    switch (_st) {
          \
      AT_PRIVATE_CASE_TYPE(at::ScalarType::Double, double, __VA_ARGS__)
          \
      AT_PRIVATE_CASE_TYPE(at::ScalarType::Float, float, __VA_ARGS__)
          \
      AT_PRIVATE_CASE_TYPE(at::ScalarType::ComplexDouble, std::complex<double>, __VA_ARGS__)
           \
      AT_PRIVATE_CASE_TYPE(at::ScalarType::ComplexFloat, std::complex<float>, __VA_ARGS__)
           \
      AT_PRIVATE_CASE_TYPE(SCALARTYPE, decltype(c10::impl::ScalarTypeToCPPType<SCALARTYPE>::t),
__VA_ARGS__)     \
      default:
           \
        AT_ERROR(#NAME, " not implemented for '", toString(_st), "'");
           \
    }
          \
  }()
```

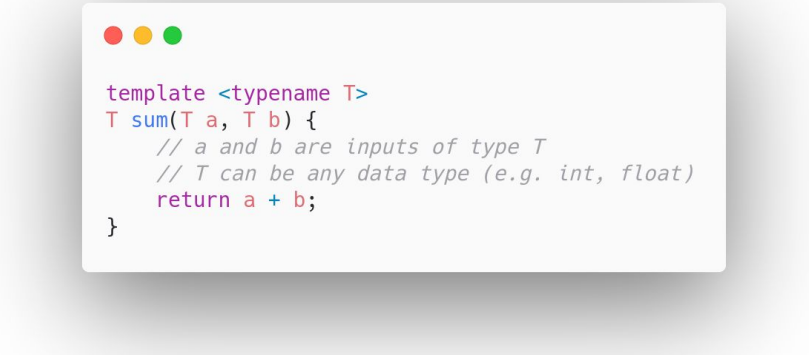
Figure 1.6. Definition of an example dispatch macro in PyTorch, *AT_DISPATCH_FLOATING_AND_COMPLEX_TYPES_AND1*. This macro is used to add data type support of floating point, complex types and given *SCALARTYPE*.



```
#define AT_PRIVATE_CASE_TYPE(enum_type, type, ...) \
  case enum_type: {                                \
    using scalar_t = type;                         \
    return __VA_ARGS__();                          \
  }
```

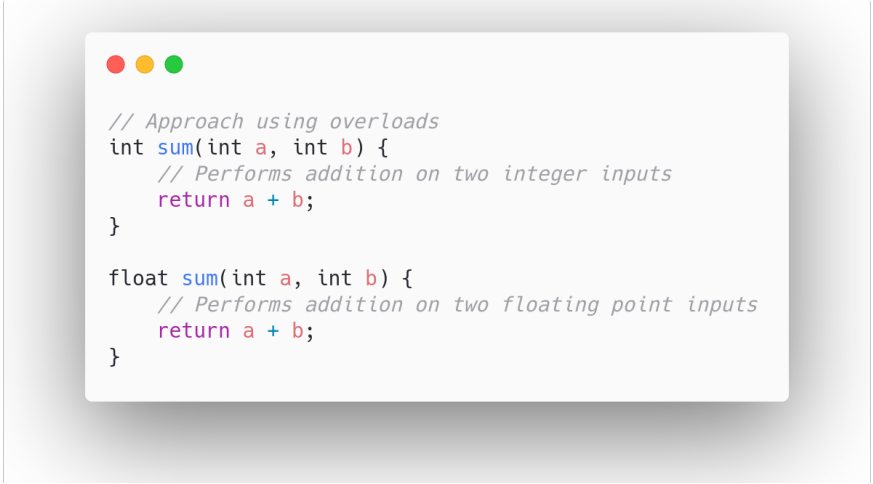
Figure 7. The macros of the form listed above use *AT_PRIVATE_CASE_TYPE*. The definition of this macro is shown here.

To better understand the functioning of these macros, this work also includes a sample of implementing a sample function *sum* using these macros. The target is to understand how a function and it's kernel is defined (for CPU devices) and how the use of macros can effectively reduce code duplication and make it more readable. The function *sum* is implemented in the given sample code. A good programming practice is to use templates while defining a function which might take different arguments.



```cpp
template <typename T>
T sum(T a, T b) {
    // a and b are inputs of type T
    // T can be any data type (e.g. int, float)
    return a + b;
}
```

Figure 1.8. Implementation of sum function using templates which eventually avoids the needs of overloads for different data types.



```cpp
// Approach using overloads
int sum(int a, int b) {
    // Performs addition on two integer inputs
    return a + b;
}

float sum(int a, int b) {
    // Performs addition on two floating point inputs
    return a + b;
}
```

Figure 1.9. Showing an example of using overloads depending on input and returning output data types. This is not preferred since templates make it easier, readable and reduces code repetition.

A kernel for sum is nothing but a way to dispatch to the correct call based on input data type. Here the macros are used to convert the output to the given data type. In case of just one input to a call of *sum*, the macros also handle those cases. While the implementation is not even close to desirable, it does the task of understanding how the macros and lambdas function in C++ and PyTorch internals. The code snippets of the header files and source files are shown in Figures 11 and 12 respectively. The code can be compiled using GNU C++ compiler (g++)[11] with the following command:

g++ src/Dispatch.cpp include/Dispatch.hpp -o out && ./out | c++filt -t

The sample output based on the given input in *src/Dispatch.cpp* is shown in Figure 1.10.

```
> ./out | c++filt -t
item: 6, type: int // 3.3f + 3.3f with output type as int
item: 7, type: double // 3.5f + 3.5f with output type as double
item: 6, type: int // 3 + 3 with output type as int
item: 7, type: int // 3.3f + 4.7f with output type as int
item: 8, type: float // 3.3f + 4.7f with output type as float
item: 8.1, type: float // 3.5f + 4.75f with output type as float
item: 8, type: double // 3.3f + 4.7f with output type as double
```

Figure 1.10. Showing output corresponding to the given inputs (in comments on each line). It's important to note here, how the values are rounded off depending on the output data type.

The sample code aims to replicate the use of these dispatch macros similar to PyTorch and a sample implementation of sum function and it's kernel. The kernel function dispatches to the correct macro depending on the given dtype. Following macros are discussed and used in the sample code:

- AT_PRIVATE_CASE_TYPE
- AT_PRIVATE_CASE_TYPE_VALUES
- AT_DISPATCH_FLOATING_TYPES
- AT_DISPATCH_VALUES_TWO

And *enums* are created for Type Promotion Strategies (output of type *int, float* or *double*) and types supported for the *dtype* argument (*int, float, double*).

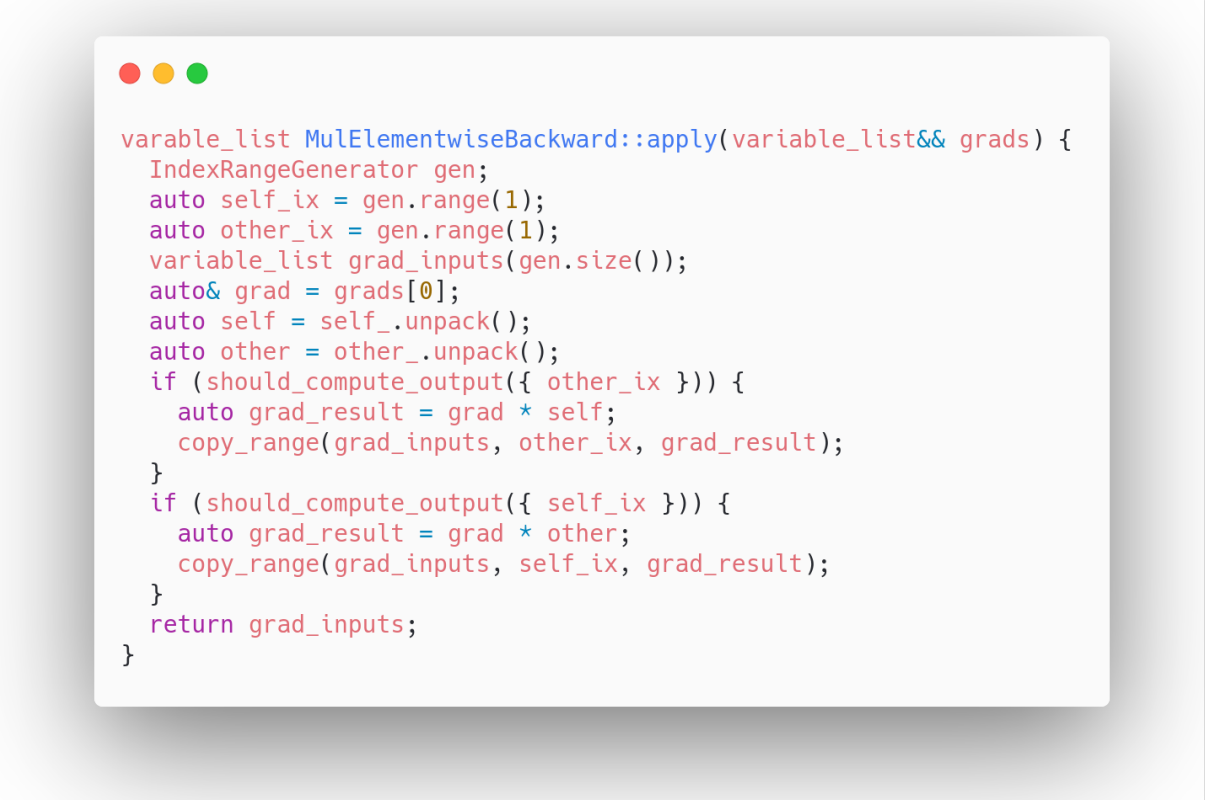
Since PyTorch also provides autograd support for each function, this also means that whenever a function is implemented, it's function to calculate gradient is explicitly defined in a file caled *derivatives.yaml* located in *torch/csrc/autograd/generated* directory of PyTorch source code. It's worth knowing how PyTorch auto generates the backward definition of the function from the registration in *derivatives.yaml* file. Taking an example of function *mul* (which performs multiplication). The auto generated function is shown in Figure 1.13.

```cpp
#include <iostream>
#include <typeinfo>

template <class T>
void _print_type(T item) {
  std::cout << "item: " << item << ", type: " << typeid(item).name() << std::endl;
}

enum TypeChangingStrategy {
  out_int, // output of type int
  out_float, // output of type float
  out_double, // output of type double
};

// types supported for dtype flag
enum Type {
  Int,
  Float,
  Double
};

#define AT_PRIVATE_CASE_TYPE(enum_type, type, value, ...) \
case enum_type:{ \
  using scalar_t = type; \
  scalar_t out = __VA_ARGS__(value); \
  _print_type(out); \
  break; \
}

#define AT_PRIVATE_CASE_TYPE_VALUES(enum_type, type, value1, value2, ...) \
case enum_type:{ \
  using scalar_t = type; \
  scalar_t out = __VA_ARGS__(scalar_t(value1), scalar_t(value2)); \
  _print_type(out); \
  break; \
}

#define AT_DISPATCH_FLOATING_TYPES(TYPE, VALUE, ...) \
[&]{ \
  const auto& the_type = TYPE; \
  auto val = VALUE; \
  switch(the_type) { \
      AT_PRIVATE_CASE_TYPE(TypeChangingStrategy::out_int, int, val, __VA_ARGS__) \
      AT_PRIVATE_CASE_TYPE(TypeChangingStrategy::out_float, float, val, __VA_ARGS__) \
      AT_PRIVATE_CASE_TYPE(TypeChangingStrategy::out_double, double, val, __VA_ARGS__) \
      default: \
            std::cout << "Error " << std::endl; \
      exit(0); \
  } \
}()

#define AT_DISPATCH_VALUES_TWO(TYPE, VALUE1, VALUE2, ...) \
[&]{ \
  const auto& the_type = TYPE; \
  auto val_1 = VALUE1; \
  auto val_2 = VALUE2; \
  switch(the_type) { \
      AT_PRIVATE_CASE_TYPE_VALUES(TypeChangingStrategy::out_int, int, val_1, val_2, __VA_ARGS__) \
      AT_PRIVATE_CASE_TYPE_VALUES(TypeChangingStrategy::out_float, float, val_1, val_2, __VA_ARGS__) \
      AT_PRIVATE_CASE_TYPE_VALUES(TypeChangingStrategy::out_double, double, val_1, val_2, __VA_ARGS__)
\
      default: \
          std::cout << "Error " << std::endl; \
      exit(0); \
  } \
}()
```

Figure 1.11. The header file defines these macros explicitly and not using PyTorch internals. *AT_DISPATCH_FLOATING_TYPES, AT_DISPATCH_VALUES_TWO, AT_PRIVATE_CASE_TYPE_VALUES* and *AT_PRIVATE_CASE_TYPE* are the macros implemented.

```cpp
#include "../include/Dispatch.h"

template <typename T>
T sum(T a, T b) {
  return a+b;
}

void sum_kernel(float f, Type dtype=Type::Int) {
  if(dtype == Type::Int) {
    AT_DISPATCH_FLOATING_TYPES(TypeChangingStrategy::out_int, f, [&](scalar_t a){
        return sum(a, a);
    });
  } else if(dtype == Type::Float) {
    AT_DISPATCH_FLOATING_TYPES(TypeChangingStrategy::out_float, f, [&](scalar_t a){
        return sum(a, a);
    });
  } else if(dtype == Type::Double) {
    AT_DISPATCH_FLOATING_TYPES(TypeChangingStrategy::out_double, f, [&](scalar_t a){
        return sum(a, a);
    });
  }
}

template <typename T>
void sum_kernel(T val1, T val2, Type dtype=Type::Int) {
  if(dtype == Type::Int) {
    AT_DISPATCH_VALUES_TWO(TypeChangingStrategy::out_int, val1, val2, [&](scalar_t a, scalar_t b) {
        return sum(a, b);
    });
  } else if(dtype == Type::Float) {
    AT_DISPATCH_VALUES_TWO(TypeChangingStrategy::out_float, val1, val2, [&](scalar_t a, scalar_t b) {
    return sum(a, b);
    });
  } else if(dtype == Type::Double) {
    AT_DISPATCH_VALUES_TWO(TypeChangingStrategy::out_double, val1, val2, [&](scalar_t a, scalar_t b) {
    return sum(a, b);
    });
  }
}

int main() {
  sum_kernel(3.3f, Type::Int);
  sum_kernel(3.5f, Type::Double);
  sum_kernel(3, Type::Int);
  sum_kernel(3.3f, 4.7f, Type::Int);
  sum_kernel(3.3f, 4.7f, Type::Float);
  sum_kernel(3.35f, 4.75f, Type::Float);
  sum_kernel(3.3f, 4.7f, Type::Double);
}
```

Figure 1.12. Implementing *sum_kernel* using the macros defined in Figure 1.11. As shown, the overloads of *sum_kernel* are defined, differing from the number of arguments. The two overloads are with one value and two values, with one *dtype* argument.



```
varable_list MulElementwiseBackward::apply(variable_list&& grads) {
  IndexRangeGenerator gen;
  auto self_ix = gen.range(1);
  auto other_ix = gen.range(1);
  variable_list grad_inputs(gen.size());
  auto& grad = grads[0];
  auto self = self_.unpack();
  auto other = other_.unpack();
  if (should_compute_output({ other_ix })) {
    auto grad_result = grad * self;
    copy_range(grad_inputs, other_ix, grad_result);
  }
  if (should_compute_output({ self_ix })) {
    auto grad_result = grad * other;
    copy_range(grad_inputs, self_ix, grad_result);
  }
  return grad_inputs;
}
```

Figure 1.13. Generated function of a custom *MulElementWise* (similar to *mul* in PyTorch). The auto *grad_result = grad * self* and *auto grad_result = grad * other* are the generated formulas. It's important to note that the *grad* variable is always placed to the left of *self* or *other* tensors. This definition will be located at *torch/csrc/autograd/generated/Functions.h*.

2.4 Brief Overview on TensorIterator API

After knowing just enough about the ATen native space of PyTorch, it will become more intuitive how and when the TensorIterator is created and understanding how TensorIterator infers the type (in which to run the operation) will become clearer. The following procedure is followed while trying to understand parts of TensorIterator API related to type promotion:

1. Create a hypothesis, after reading the TensorIterator code on where is the type inferred for the input and output tensors.
2. Write a sample code to create a tensor, and use *gdb* to back trace.
3. Observe where the TensorIterator is created and where the type is inferred.
4. Validate the hypothesis and make conclusions.

Observing the TensorIterator code, the author had assumed the type is inferred in *at::TensorIterator::compute_types()* function in *TensorIterator.cpp*. The function assumably computes the data type of the operand (or tensor). To validate the hypothesis and make

corrections if any, *gdb* debugger comes into the frame here. Code shown in Figure 1.14 is back traced.

```
# Using device as cuda as we are doing everything to make sure PyTorch runs seamlessly on our GPUs
sample_tensor = torch.tensor([1, 2, 3], dtype=torch.float, device="cuda")
```

Figure 1.14. Sample code to create an input tensor containing *[1, 2, 3]* elements as *float32* format and on CUDA device. Most of the experiments in the work are run on CUDA devices, as the primary purpose is to improve performance of PyTorch on GPUs.

The code above is passed to *gdb* debugger and back traced. The output is given in the table below.

1	*torch::autograd::THPVariable_tensor(_object*, _object*, _object*) ()* Location: torch/csrc/autograd/generated/python_torch_functions.cpp torch/csrc/autograd/generated/... `static PyObject * THPVariable_tensor(PyObject* self, PyObject* args, PyObject* kwargs) {` `  // …` `  return THPVariable_Wrap(torch::utils::tensor_ctor(torch::tensors::get_default_dispatch_key(), torch::tensors::get_default_scalar_type(), args, kwargs));` `  // ..` `}` Figure 1.15. Implementation of *THVariable_tensor* function which binds the python function to the correct dispatch call in C++.
2	*torch::utils::tensor_ctor(c10::DispatchKey, c10::ScalarType, _object*, _object*) ()* Location: torch/csrc/utils/tensor_new.cpp Note: This creates another tensor

torch/csrc/utils/tensor_new.cpp

```
Tensor tensor_ctor(c10::DispatchKey dispatch_key, at::ScalarType scalar_type, PyObject* args, PyObject*
kwargs) {
  static PythonArgParser parser({
    "tensor(PyObject* data, *, ScalarType dtype=None, Device? device=None, bool pin_memory=False, bool
requires_grad=False, DimnameList? names=None)",
  });

  constexpr int ctor_num_args = 6;
  ParsedArgs<ctor_num_args> parsed_args;
  auto r = parser.parse(args, kwargs, parsed_args);
  if (r.idx == 0) {
    PyObject* data = r.pyobject(0);
    if (THPVariable_Check(data)) {
      PyErr_WarnEx(PyExc_UserWarning,
        "To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() "
        "or sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).", 1);
    }

    bool type_inference = r.isNone(1);
    bool pin_memory = r.toBool(3);
    bool args_requires_grad = r.toBool(4);
    auto new_tensor = internal_new_from_data(
                typeIdWithDefault(r, 2, dispatch_key),
                r.scalartypeWithDefault(1, scalar_type),
                r.deviceOptional(2),
                data,
                true,
                true,
                type_inference,
                pin_memory);
    auto names = r.toDimnameListOptional(5);
    if (names) {
      at::namedinference::propagate_names(new_tensor, *names, /*validate_names=*/true);
    }
    new_tensor.detach_(); // ensure new_tensor a leaf node
    new_tensor.set_requires_grad(args_requires_grad);
    return new_tensor;
  }
  throw std::runtime_error("tensor(): invalid arguments");
}
```

Figure 1.16. The function is responsible for creating a new tensor from the input tensor passed to the Python function call. The code in bold is responsible for creating a new tensor and setting *requires_grad* to *True* if passed in the function call.

3	*torch::utils::(anonymous namespace)::internal_new_from_data(c10::DispatchKey, c10::ScalarType, c10::optional<c10::Device>, _object*, bool, bool, bool, bool) ()* Location: *torch/csrc/utils/tensor_new.cpp* Signature: *Tensor internal_new_from_data(c10::DispatchKey dispatch_key, at::ScalarType scalar_type, c10::optional<Device> device_opt, PyObject* data, bool copy_variables, bool copy_numpy, bool type_inference, bool pin_memory = false)* Note: This function is called from *tensor_ctor* in the 2nd step, and all the boolean arguments (*copy_variables, copy_numpy, type_inference*) are explicitly set to *true*.
4	*at::Tensor::to(c10::Device, c10::ScalarType, bool, bool, c10::optional<c10::MemoryFormat>) const ()* Location: *build/aten/src/ATen/core/TensorMethods.h*

	Signature: *inline Tensor Tensor::to(Device device, ScalarType dtype, bool non_blocking, bool copy, c10::optional<MemoryFormat> memory_format) const* This is called to convert the created variable (pointer to *THPVariable* object) to the inferred scalar type, and device.
5	*c10::detail::wrap_kernel_functor_unboxed_<c10::detail::WrapRuntimeKernelFunctor_<at::Tensor (*)(at::Tensor const&, c10::Device, c10::ScalarType, bool, bool, c10::optional<c10::MemoryFormat>), at::Tensor, c10::guts::typelist::typelist<at::Tensor const&, c10::Device, c10::ScalarType, bool, bool, c10::optional<c10::MemoryFormat> > >, at::Tensor (at::Tensor const&, c10::Device, c10::ScalarType, bool, bool, c10::optional<c10::MemoryFormat>)>::call(c10::OperatorKernel*, at::Tensor const&, c10::Device, c10::ScalarType, bool, bool, c10::optional<c10::MemoryFormat>) ()* Location: Irrelevant here. This appears in the back trace as the debug symbols are also shown.
6	*torch::autograd::VariableType::(anonymous namespace)::to(at::Tensor const&, c10::Device, c10::ScalarType, bool, bool, c10::optional<c10::MemoryFormat>) ()* Location: *torch/csrc/autograd/generated/VariableTypeEverything.cpp* Signature: *Tensor to(const Tensor & self, const TensorOptions & options, bool non_blocking, bool copy, c10::optional<MemoryFormat> memory_format)*
7	*at::TypeDefault::to(at::Tensor const&, c10::Device, c10::ScalarType, bool, bool, c10::optional<c10::MemoryFormat>) ()* Location: *build/aten/src/ATen/TypeDefault.cpp*
8	*at::native::to(at::Tensor const&, c10::Device, c10::ScalarType, bool, bool, c10::optional<c10::MemoryFormat>) ()* Location: *aten/src/ATen/native/TensorConversions.cpp*
9	*at::native::to_impl(at::Tensor const&, c10::TensorOptions const&, bool, bool, c10::optional<c10::MemoryFormat>) ()* Location: *aten/src/ATen/native/TensorConversions.cpp*

	Note: This essentially creates a copy of tensor with given options.
10	*at::Tensor::copy_(at::Tensor const&, bool) const ()* Location: build/aten/src/ATen/core/TensorMethods.h Note: This is surely a dynamic dispatch to *aten::copy_*. Check *torch/csrc/autograd/generated/RegistrationDeclarations.h* - *copy_* function is registered with *aten::copy_*.
11	*torch::autograd::VariableType::(anonymous namespace)::copy_(at::Tensor&, at::Tensor const&, bool) ()*
12	*at::native::copy_(at::Tensor&, at::Tensor const&, bool) ()* Location: aten/src/ATen/native/Copy.cpp
13	*at::native::copy_impl(at::Tensor&, at::Tensor const& bool) ()* This is where the TensorIterator object is created. aten/src/ATen/native/Copy.cpp ``` static Tensor & copy_impl(Tensor & self, const Tensor & src, bool non_blocking) { // ... auto iter = TensorIterator(); iter.set_check_mem_overlap(true); iter.add_output(self); iter.add_input(src); iter.dont_resize_outputs(); iter.dont_compute_common_dtype(); iter.build(); if (iter.numel() == 0) { return self; } DeviceType device_type = iter.device_type(0); if (iter.device_type(1) == kCUDA) { device_type = kCUDA; } else if (iter.device_type(1) == kHIP) { device_type = kHIP; } // TODO: if we need to, we can also enable this path for quantized tensor if (device_type == kCPU && copy_transpose_valid(self, src) && !self.is_quantized()) { copy_same_type_transpose_(self, src); return self; } copy_stub(device_type, iter, non_blocking); return self; } ``` Figure 1.17. The code in bold is the place where the tensor iterator is built and its properties are set.

14	*at::TensorIterator::build() ()* Location: *aten/src/ATen/native/TensorIterator.cpp* aten/src/ATen/native/TensorIterator.cpp Figure 1.18. Definition of method function *build* of the *TensorIterator* API. This is the part where the *TensorIterator* is built from the input and the output tensors. It also sets the relevant characteristics like number of outputs, type promotion (if any) and more.
15	*at::TensorIterator::compute_types()* Location: *aten/src/ATen/native/TensorIterator.cpp*

```cpp
void TensorIterator::build() {
  // check input tensors memory format to use it during output allocation
  analyze_memory_format();
  // set is_output and is_read_write flags on appropriate tensors
  mark_outputs();
  // Check that the outputs have no internal overlap
  // and do not share memory with inputs.
  check_mem_overlaps();
  // Check that input dimensions are aligned correctly & compute outnames.
  compute_names();
  // compute the broadcasted shape
  compute_shape();
  // compute the result dtype and device
  compute_types();
  // try fast setup output tensor, if failed, fallback to normal setup
  if (!fast_set_up()) {
    // compute each tensor's stride after broadcasting
    compute_strides();
    // re-order dimensions to improve coalescing
    reorder_dimensions();
    // allocate the output tensor if it's not provided
    allocate_outputs();
    // coalesce adjacent dimensions when possible
    coalesce_dimensions();
  }
  // perform name inference
  propagate_names_to_outputs();

  for (auto& op : operands_) {
    TORCH_INTERNAL_ASSERT(op.tensor.defined());
    op.data = op.tensor.data_ptr();
  }

  // zero out offsets
  // If the tensor is a scalar, we leave room for it
  // So index translations in reduction can access
  // a valid value for the offset
  int64_t ndim_offsets = (ndim() ? ndim() : 1);
  view_offsets_ = DimVector(ndim_offsets, 0);
}
```

	Note: This is where the common data type from input and output tensors is inferred, depending on the type promotion strategy.

Table 2.1: Table showing the output of gdb back tracking command to understand where the *TensorIterator* computes types.

As it's visible from the table above, the last function we reached is *at::TensorIterator::compute_types* which also indicates the place where TensorIterator actually attempts to infer data types of output and input tensors. The hypothesis made initially has hence been validated.

CHAPTER 3

Understanding the TensorIterator API and underlying type promotion logic

Preface

This chapter revolves around the *TensorIterator* API of PyTorch and talks about the type promotion rules for unary functions in PyTorch. The *TensorIterator* API of PyTorch is complicated and thus, this chapter attempts to take a step by step approach to explain type promotion and how and where the *TensorIterator* is built during the call of a unary function.

3.1 Introduction

This aim of this chapter is to describe the TensorIterator API of PyTorch and how the type promotion strategy is implemented. Since most of the work is code, the code snippets are also shown here in order to better explain the functionality. The code for TensorIterator is located in the *aten/src/ATen/native* and it's the place where most of the work is centered. As per the conventions followed by any library written in C++, the code is divided into a header file and a source file.

- *TensorIterator.cpp*[12] - All of the source code, related to building a TensorIterator from input and output tensors resides here.
- *TensorIterator.h*[12] - This header file contains the declarations of the functions defined in the source file.

TensorIterator API builds a TensorIterator object using a *build* function defined in *aten/src/ATen/native/TensorIterator.cpp*. The properties of the *TensorIterator* object are set in the function depending on the type of operation. For example, if the operation is unary operation (that is, takes only one input Tensor), then the TensorIterator object is built, and it's properties are set in *TensorIterator::unary_op* method of the *TensorIterator* class. The definition of *unary_op* method is shown in the Figure 2.1 below. It defines the tensor iterator building method for a unary op in PyTorch.

3.2 Understanding Relationship between TensorIterator and UnaryOps

The function *unary_op* is called from the file *UnaryOps.cpp*[13], from one of the helper functions *unary_op_out_impl* and its derivatives. The helper functions are meant to help nearly all the unary functions to call *TensorIterator::unary_op* and the corresponding kernel function.

aten/src/ATen/native/TensorIterator.cpp

```
TensorIterator TensorIterator::unary_op(Tensor& out, const Tensor& a,
    bool check_mem_overlap) {
  auto iter = TensorIterator();
  iter.set_check_mem_overlap(check_mem_overlap);
  iter.add_output(out);
  iter.add_input(a);
  iter.num_outputs_ = 1;
  iter.build();
  return iter;
}
```

Figure 2.1. *TenorIterator::unary_op* method builds the TensorIterator object from the input and output tensors, and sets its properties.

Before the function *unary_op* is discussed line by line, it's important to be aware where the function *unary_op* is called. The call is mentioned in bold in the Figure 2.2 below.

```
template <typename Stub>
static inline Tensor& unary_op_impl_out(Tensor& result, const Tensor& self, Stub& stub) {
  auto iter = TensorIterator::unary_op(result, self,
    /*check_mem_overlap=*/true);
  stub(iter.device_type(), iter);
  return result;
}
```

Figure 2.2. Function *unary_op_impl_out* is a helper function to reduce usage of any redundant function in the call of each operation. Each unary function calls this helper function ultimately which creates a *TensorIterator* object and is then passed to the kernel (*stub*).

The method function *unary_op* is at the end, in focus where the output tensor is added to the *TensorIterator* API (using *iter.add_output(out)*), input tensor is added to the *TensorIterator* API (using *iter.add_input(a)*), number of outputs are set to 1 (using *iter.num_outputs_ = 1*) and the iterator object is built using *iter.build()*. The method function *build* is explained in the

Figure 2.3 below. The function *build* is responsible to check memory format, compute strides, common data type for input and output tensors (depending on the type promotion strategy), compute shape, copy input tensor values to the operand values of the *TensorIterator* and more.

3.3 Type Computation in TensorIterator API

If taken a look at *compute_types* function, it is responsible for choosing the type promotion strategy as well, which is highly relevant for the task. Comparison with how binary operations do the type promotion strategy is useful here, since one of the main contributors at Facebook (PyTorch) had recently enabled type promotion for Binary Operations before the start of this project. The *binary_op* is given in the Figure 2.3 below.

aten/src/ATen/native/TensorIterator.cpp

```
TensorIterator TensorIterator::binary_op(Tensor& out, const Tensor& a,
    const Tensor& b, bool check_mem_overlap) {
  auto iter = TensorIterator();
  iter.set_check_mem_overlap(check_mem_overlap);
  iter.add_output(out);
  iter.add_input(a);
  iter.add_input(b);
  iter.allow_cpu_scalars_ = true;
  iter.promote_common_dtype();
  iter.build();
  return iter;
}
```

Figure 2.3. Definition of *binary_op* situated in *TensorIterator.cpp* file. The difference from *unary_op* shown in Figure 2.1 is shown in bold, which sets the type promotion strategy available in the TensorIterator API.

aten/src/ATen/native/TensorIterator.cpp

```
void TensorIterator::build() {
  // check input tensors memory format to use it during output allocation
  analyze_memory_format();
  // set is_output and is_read_write flags on appropriate tensors
  mark_outputs();
  // Check that the outputs have no internal overlap
  // and do not share memory with inputs.
  check_mem_overlaps();
  // Check that input dimensions are aligned correctly & compute outnames.
  compute_names();
  // compute the broadcasted shape
  compute_shape();
  // compute the result dtype and device
  compute_types();
  // try fast setup output tensor, if failed, fallback to normal setup
  if (!fast_set_up()) {
    // compute each tensor's stride after broadcasting
    compute_strides();
    // re-order dimensions to improve coalescing
    reorder_dimensions();
    // allocate the output tensor if it's not provided
    allocate_outputs();
    // coalesce adjacent dimensions when possible
    coalesce_dimensions();
  }
  // perform name inference
  propagate_names_to_outputs();

  for (auto& op : operands_) {
    TORCH_INTERNAL_ASSERT(op.tensor.defined());
    op.data = op.tensor.data_ptr();
  }

  // zero out offsets
  // If the tensor is a scalar, we leave room for it
  // So index translations in reduction can access
  // a valid value for the offset
  int64_t ndim_offsets = (ndim() ? ndim() : 1);
  view_offsets_ = DimVector(ndim_offsets, 0);
}
```

Figure 2.4. The method function *build()* checks memory format, computes the result data type and device, performs name inference, computes tensor strides after broadcasting and zeros out offsets. The relevant function here is only *compute_types* which computes the result data type and device.

While the requirement from the NVIDIA side was limited to supporting explicit *dtype* argument which could eventually streamline their interaction with autocasting and thus preventing the need to explicitly precast the output data type using a separate kernel. The PyTorch maintainers though had a preference of having implicit type promotion matching

NumPy before adding explicit type promotion behavior. The rest of this chapter is focused on explaining the use of *PROMOTE* type promotion strategy in computing common data type and output data type in the *TensorIterator* API.

aten/src/ATen/native/TensorIterator.h

```
enum class CommonDTypeStrategy : uint8_t {
  NONE,
  // Do not compute a common dtype
  CHECK,
  // Compute and validate a common dtype but don't promote.
  PROMOTE_INPUTS,
  // Promote common dtype but only validate inputs
  // (comparison ops have boolean output)
  PROMOTE // Promote to common dtype.
};

void promote_common_dtype() {
  common_dtype_strategy_ = CommonDTypeStrategy::PROMOTE;
}
```

Figure 2.5. There are in total 4 common type promotion strategies as listed in the *enum class CommonDTypeStrategy* which include *NONE, CHECK, PROMOTE_INPUTS* and *PROMOTE*. The function *promote_common_dtype* sets the strategy to *PROMOTE*.

aten/src/ATen/native/TensorIterator.cpp

```
void TensorIterator::compute_types() {
  // ...
  if (common_dtype_strategy_ == CommonDTypeStrategy::PROMOTE_INPUTS) {
    TORCH_CHECK(!missing_output_dtypes, "unable to compute and promote common dtype based only on
inputs if there are missing dtypes for outputs");
  }

  bool compute_common_dtype = (common_dtype_strategy_ != CommonDTypeStrategy::NONE);
  bool compute_common_dtype_only_for_inputs = (common_dtype_strategy_ ==
CommonDTypeStrategy::PROMOTE_INPUTS);
  // ...
}
```

Figure 2.6. Boolean variables *compute_common_dtype* is set to *true* and *compute_common_dtype_only_for_inputs* is set to *false* in case of *PROMOTE* type strategy.

The variable *compute_common_dtype* is set to *true* if the strategy is set to anything except *NONE*. That is, if the requirement is to promote the data type of input or output tensors then the *compute_common_dtype* flag is set to *true*. The type promotion strategy *PROMOTE_INPUTS* does not promote the output data type to the common data type inferred from inputs and only validates input data type, and promotes to the common data type.

aten/src/ATen/native/TensorIterator.cpp

```
void TensorIterator::compute_types() {
  // ...
  else {
    op.device = common_device;
    if (compute_common_dtype_only_for_inputs && op.is_output) {
      op.target_dtype = op.current_dtype;
    } else {
      op.target_dtype = common_dtype_;
    }
  }
  // ...
}
```

Figure 2.7. The part of code in *compute_types* method function, where if the operand is an output tensor and the strategy is set to *PROMOTE_INPUTS*, then the target data type is set to current data type (that is the data type of output tensor) else it is set to the common data type inferred (*commom_dtype_*) which is discussed next.

The common data type calculation is documented well in the PyTorch source code. The function *compute_common_type_* does the common data type calculation from the given operands (*at::ArrayRef<OperandInfo> operands*). These operands are stored as objects of the class *OperandInfo* defined in the *TensorIterator*.

As shown in Figure 2.8, in case when the data types of input and output tensors are not same, then *update_result_type_state* is held responsible to iteratively infer the common data type. Initially, it compares *ScalarType::Undefined* with the data type of the first input tensor (or operand) and updates that to the *state* variable. In the next iteration, the first input tensor's data type and the next input tensor's data type (if any, else output tensor's data type) are compared using the type promotion table shown in Figure 3.7. The final state is then wrapped, and returned as a tuple. This can be summarized as follows.

1. If all tensors/operands are of the same type, return the data type as the common data type.
2. If all tensors/operands are not of the same type:
    a. Compare each tensor's data type with the next one in the order, using the table given in Figure 3.7. If any one of them is *Undefined*, use the tensor's data type which is defined.
    b. Continue the iteration till the last tensor and return the final state as a tuple.

aten/src/ATen/native/TensorIterator.cpp

```
static std::tuple<Device, ScalarType, bool> compute_common_type_(at::ArrayRef<OperandInfo> operands) {
  // See [Result type computation] in TensorIterator.h
  auto device = compute_device(operands);
  auto common_type = ScalarType::Undefined;
  bool all_same_type = true;
  for (const auto& op: operands){
    if (!op.tensor.defined()) continue;
    //don't handle scalars
    if (op.tensor.dim() > 0){
      ScalarType current = op.current_dtype;
      if (current == ScalarType::Undefined){
        all_same_type = false;
        break;
      }
      if (common_type == ScalarType::Undefined) common_type = current;
      if (common_type != current) {
        all_same_type = false;
        break;
      }
    } else {
      all_same_type = false;
      break;
    }
  }
  if (all_same_type) {
    return std::make_tuple(device, common_type, true);
  }

  at::native::ResultTypeState state = {};
  for (const auto& op : operands) {
    state = at::native::update_result_type_state(op.tensor, state);
  }
  auto dtype = at::native::result_type(state);

  auto result = std::make_tuple(device, dtype, false);
  TORCH_INTERNAL_ASSERT(dtype != ScalarType::Undefined);
  return result;
}
```

Figure 2.8. The method function *compute_common_type_* computes the common data type from the operands (input and output tensors). This strategy is hidden in the call of *update_result_type_state*.

As it's clear, using *PROMOTE* strategy similar to what has been done with Binary Ops in PyTorch made the best sense for the use case scenario.

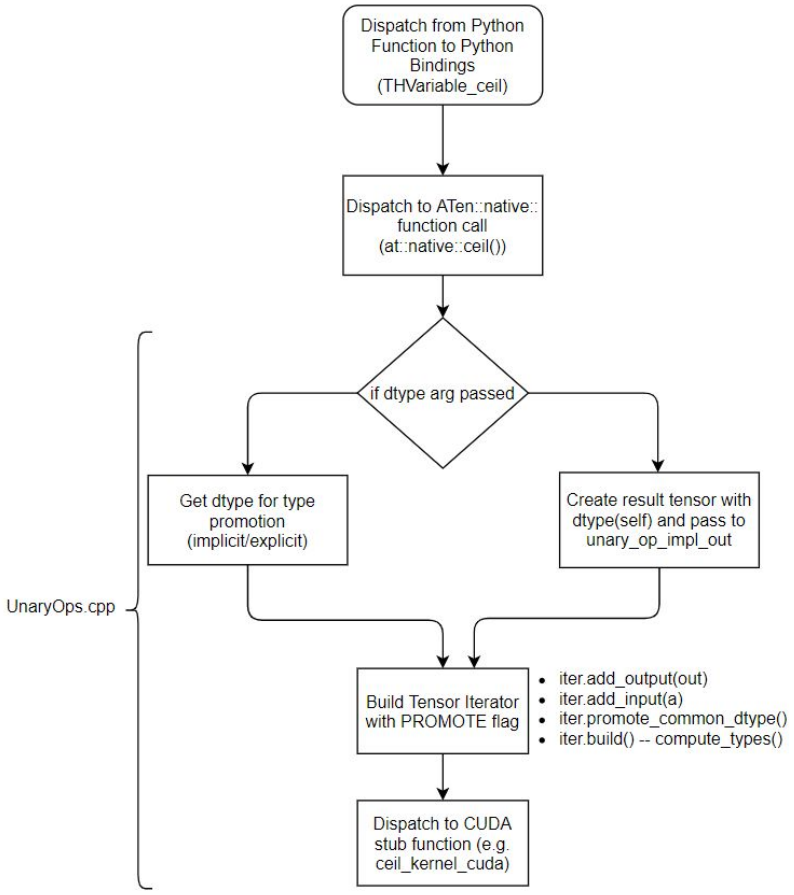


Figure 2.9. Part of type promotion and dispatch in *UnaryOps.cpp* file.

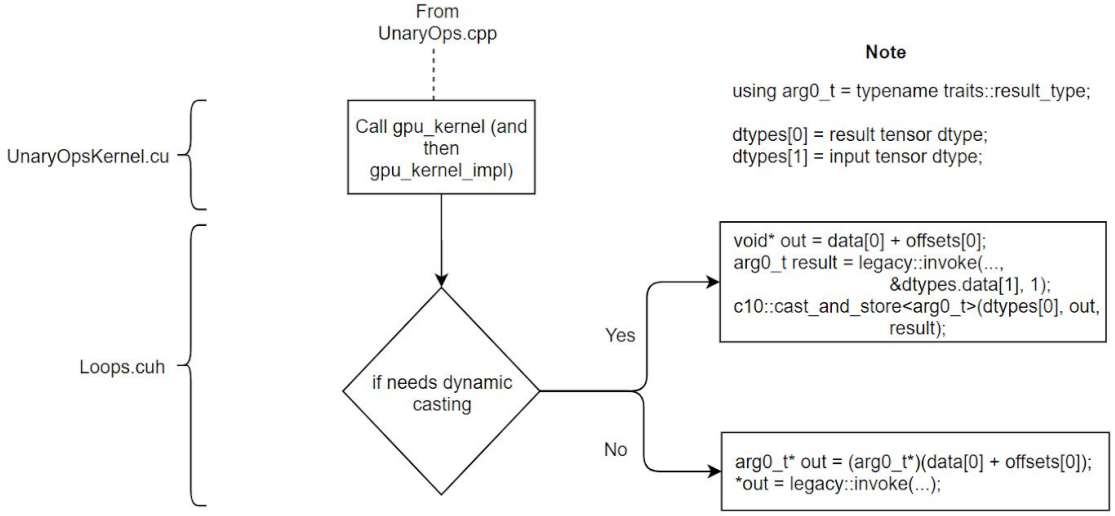


Figure 2.10. Flow chart in continuation to Figure 2.9 showing the usage of *cast_and_store* function. *dtypes[0]* is the input tensor data type and *dtypes[1]* is the output tensor data type.

CHAPTER 4

Adding Type Promotion Support for Unary Floating Universal Functions in PyTorch: Implementation

Preface

This chapter discusses the implementation details of enabling type promotion in PyTorch for the relevant Unary Functions. It also discusses the approach taken to understand which function should be promoted and which should be untouched. The comparison is discussed between JAX and NumPy as well, and the scripts to make the comparisons are also discussed. By the end of this chapter, the implementation details will be combined and the results will be shown.

4.1 Introduction

Based on the discussion with the upstream (PyTorch maintainers from Facebook), the decision was made to keep the type promotion limited to floating unary functions. Floating Unary Functions are the ops (or functions) in PyTorch which only require one input tensor, and promote to default floating type.

aten/src/ATen/native/TensorIterator.cpp

```
TensorIterator TensorIterator::unary_floating_ufunc(Tensor& out, const Tensor& a,
                                                    bool check_memory_overlap) {
  auto iter = TensorIterator();
  iter.set_check_mem_overlap(check_mem_overlap);
  iter.add_output(out);
  iter.add_input(a);
  iter.num_outputs_ = 1;
  iter.promote_common_dtype();
  iter.build();
  return iter;
}
```

Figure 3.1. A function similar to *TensorIterator::unary_op* was implemented, named as *TensorIterator::unary_floating_ufunc* which promotes the data type to default floating type internally if no output data type is mentioned explicitly.

The type promotion rules are followed based on the type promotion rules in PyTorch which are inherited from NumPy. These rules are highly discussed upon, thought about and then used in a library like PyTorch. It's worth understanding how NumPy promotes it's unary universal functions which is briefly described in Table 3 and the comparison is further

detailed in section 4.2.

Category of Type Promotion	Promotion Rule	Ops following the mentioned Category
Type 0	No Upcasting	*abs, real, imag, bitwise_not, clip, neg*
Type 1	int8 - float16, int16 - float32, int32 - float64, int64 - float64, bool - float16	*ceil, expm1, floor, log, log10, log1p, log2, sin, sinh, sqrt, trunc, atan, cos, tan*
Type 2	(int8, int16, int32, int64, bool) - float64	*angle*
Type 3	bool - int8	*conjugate, reciprocal, square*
Type 4	bool - float16	*round*

Table 3.1: Type Promotion (Upcasting) rules in NumPy for Unary Universal Functions. These rules are latest by 20th February 2020 and could have been changed past the given date. There are in total 5 categories of type promotion, of which Type 0 represents the no-upcasting category. The third column corresponds to the list of unary universal functions following the given category of type promotion.

Though PyTorch attempts to be consistent with the existing approach taken by NumPy, it's conscious and thus, it was preferred to also discuss type promotion approach in JAX library. This is briefly discussed in Section 4.2 and 4.3.

4.2 Type Promotion in NumPy and JAX for Unary Ops

The discussion with one of the PyTorch maintainers from the Facebook team is given in the Figure below. The intention is to understand whether or not PyTorch needs to mimic the rules of NumPy.

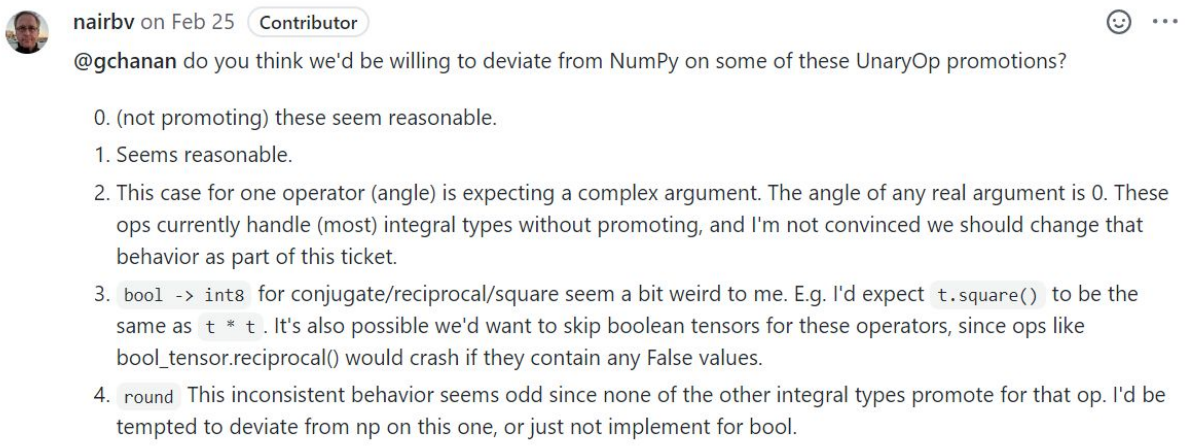


Figure 3.2. A screenshot of the discussion from the pull request opened to enable type promotion in PyTorch. Brian Vaughn is one of the developers who has dealt with the type promotion issues and pull requests lately.

It's important to note that NumPy doesn't have all the ops available in PyTorch. A comprehensive list of ops not available in NumPy is given below.

1. *np.acos* (treated as *np.arccos*)
2. *np.asin* (treated as *np.arcsin)*
3. *np.bitwise_not* (treated as *np.invert*)
4. *np.frac* (treated as *np.modef*)
5. *np.digamma* (preferred alternate: scipy.special.digamma)
6. *np.sigmoid* (not present)
7. *np.neg* (treated as *np.negative*)
8. *np.clamp* (treated as *np.clip*)
9. *np.polygamma* (preferred alternate: *scipy.special.polygamma*)
10. *np.mvlgamma* (not present)

As mentioned above, some of the ops in NumPy don't necessarily use the same keyword as in PyTorch while a few are not present (*sigmoid*, *mvlgamma*, *polygamma*, *digamma*). If there is any preferred alternative to these ops in any other library, then the promotion rule of that op is discussed, else the mathematical formula is discussed and thought upon.

After discussion with the upstream and the author's mentor (Michael Carilli), it was decided to take a look at the list of ops in JAX and how the library promotes integral tensors.

- Type AllToBool (promote all input data types to boolean for output tensors)
    - *logical_not, isfinite, isinf, isnan*
- Type AllSame but int64 to int32 (keep the data type same except for *int64* input type)

    - *imag, neg, abs, sign, square, bitwise_not*
- Type AllSame (do not perform any implicit type promotion)
    - *real, conj, round*
- Type AllToFloat32 (promote all integral and boolean tensors to *float32*)
    - *angle, exp, log, log2, log10, expm1, log1p, sqrt, reciprocal, sin, cos, tan, arccos, arctan, sinh, cosh, tanh, floor, ceil*

Functions not yet implemented in *jax.numpy* are: *np.trunc* and *np.modf* (similar to *torch.frac*). While functions not present in NumPy/jax.numpy: *erf, erfc, erfinv, rsqrt, sigmoid, digamma, lgamma, polygamma, mvlgamma, clamp*.

Doing this comparison motivated the upstream to stick to promoting only integral or boolean input tensors to default floating type (by default *float32* unless set otherwise). These ops which supported this conversion, are thus termed as unary floating universal functions.

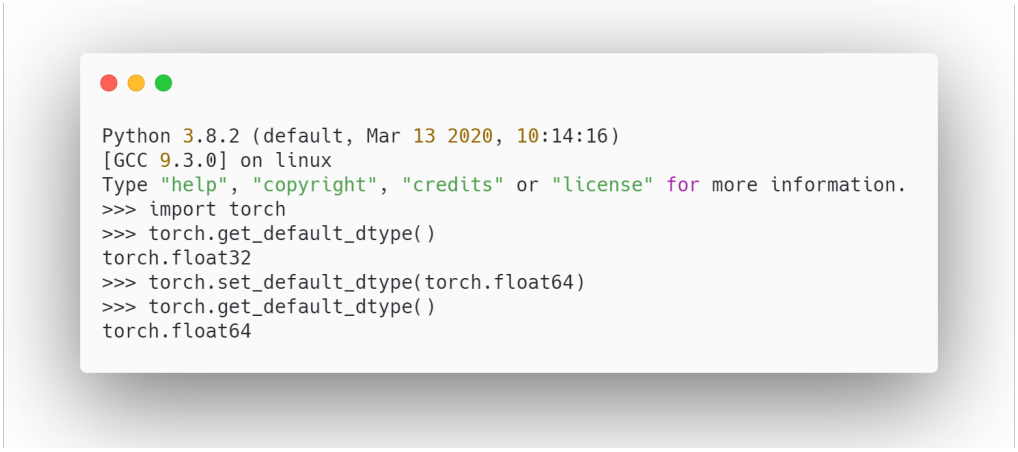


Figure 3.3. Figure depicting usage of *get_default_dtype* and *set_default_dtype* in PyTorch.

Enabling type promotion will help users to operate the relevant unary operation in PyTorch on an integer tensor and be given a floating point output (the output data type is the default floating type). In PyTorch, by default the default floating type is *float32* and can be set using *torch.set_default_dtype()* as shown in Figure 3.3. The default floating type can be changed to *float64* or *float16* depending on the requirements.

4.3 Implementing Type Promotion for Unary Floating Universal Functions

This section covers the implementation details of type promotion using *PROMOTE* strategy in PyTorch for Unary Floating Universal Functions. As mentioned before, the aim is to promote integral and boolean tensors to default floating type. Figure 3.4 helps understand how the default floating type is computed and set during a PyTorch session.

Python Shell

```
>>> x = torch.tensor(2, device="cuda")
>>> y = torch.tensor(False, device="cuda")

>>> torch.abs(x).dtype # No change
torch.int64
>>> torch.ceil(x).dtype # Type 1 (int64 to float64)
torch.float64
>>> torch.angle(x).dtype # Type 2 (int64 to float64)
torch.float64
>>> torch.conj(y).dtype # Type 3 (bool to int8)
torch.int8
>>> torch.round(y).dtype # Type 4 (bool to fp16 if cuda, else fp32)
torch.float16
```

Figure 3.4. A python shell showing how the types are promoted implicitly for each category given in table 3.

As shown in the figure above, two sample tensors *x* and *y* are created on a CUDA device. *x* is an integer tensor while *y* is a boolean tensor. Originally, without type promotion, doing *torch.angle(x)* will return an error on giving an integer input, but now it will just work fine because it implicitly promotes an *int64* type to *float64* type. Most of the unary functions in PyTorch use helper functions defined in *aten/src/ATen/native/UnaryOps.cpp* file which helps calling the stub (appropriate kernel for the given function). One example is given in the figure below of *acos*.

aten/src/ATen/native/UnaryOps.cpp

```
Tensor& acos_out(Tensor& result, const Tensor& self) {
  return unary_floating_ufunc_op_impl_out(result, self, acos_stub);
}
Tensor acos(const Tensor& self) { return unary_floating_ufunc_op_impl(self, at::acos_out); }
Tensor& acos_(Tensor& self) { return unary_floating_ufunc_op_impl_(self, at::acos_out); }
```

Figure 3.5. Example implementation of a unary function *acos* which takes input tensors and calls the kernel function of the corresponding op and builds a *TensorIterator* using the type promotion rule.

The helper functions *unary_floating_ufunc_op_impl*, *unary_floating_ufunc_op_impl_*, and *unary_floating_ufunc_op_impl_out* are shown in the figure below.

aten/src/ATen/native/UnaryOps.cpp

```
// unary_floating_ufunc_* functions are helper functions to support integral and bool Tensors
// to default floating type promotion for floating ufuncs. For non-floating ufuncs, use existing
// helper functions (unary_op_impl*) if needed.
template <typename Stub>
static inline Tensor& unary_floating_ufunc_op_impl_out(Tensor& result, const Tensor& self, Stub& stub)
{
  TORCH_CHECK(!isIntegralType(result.scalar_type(), /*includeBool=*/ true),
    "Attempted to call a function that returns a tensor of floating dtype ",
    "but requested a tensor of bool or integral dtype. ");

  auto iter = TensorIterator::unary_floating_ufunc(result, self,
    /*check_mem_overlap=*/true);
  stub(iter.device_type(), iter);
  return result;
}

template <typename OutImpl>
static inline Tensor unary_floating_ufunc_op_impl(const Tensor& self, OutImpl& out_impl) {
  if (isIntegralType(self.scalar_type(), /*includeBool=*/ true)) {
    const auto scalar_type = typeMetaToScalarType(c10::get_default_dtype());
    Tensor result = at::empty({0}, self.options().dtype(scalar_type));
    return out_impl(result, self);
  }

  Tensor result = at::empty({0}, self.options());
  return out_impl(result, self);
}

template <typename OutImpl>
static inline Tensor& unary_floating_ufunc_op_impl_(Tensor& self, OutImpl& out_impl) {
  TORCH_CHECK(!isIntegralType(self.scalar_type(), /*includeBool=*/ true),
    "Attempted to call a function that returns a tensor of floating dtype ",
    "inplace on a tensor of bool or integral dtype. ",
    "Perform the operation out-of-place ",
    "or cast the tensor to a floating dtype first.");
  return out_impl(self, self);
}
```

Figure 3.6. Implementation of the helper functions for Unary Universal Functions in PyTorch. These helper functions check if the input tensor data type is either integral or boolean, if yes the type is promoted according to the rules embedded in the *TensorIterator* logic.

The clue is passed to the type promotion rules by passing output data type as the default floating type by using scalar type (from *typeMetaToScalarType(c10::get_default_dtype())* in Figure 3.6) as the data type of output tensor. The function *get_default_dtype()* returns the default floating point data type in Pytorch session. The type promotion rules are given below.

c10/core/ScalarType.h

```
static inline ScalarTypee promoteTypes(ScalarType a, ScalarType b) {
  // This is generated according to NumPy's promote_types
  constexpr auto u1 = ScalarType::Byte;
  constexpr auto i1 = ScalarType::Char;
  constexpr auto i2 = ScalarType::Short;
  constexpr auto i4 = ScalarType::Int;
  constexpr auto i8 = ScalarType::Long;
  constexpr auto f2 = ScalarType::Half;
  constexpr auto f4 = ScalarType::Float;
  constexpr auto f8 = ScalarType::Double;
  constexpr auto c2 = ScalarType::ComplexHalf;
  constexpr auto c4 = ScalarType::ComplexFloat;
  constexpr auto c8 = ScalarType::ComplexDouble;
  constexpr auto b1 = ScalarType::Bool;
  constexpr auto bf = ScalarType::BFloat16;
  constexpr auto ud = ScalarType::Undefined;

  static constexpr ScalarType _promoteTypesLookup[static_cast<int>(
      ScalarType::NumOptions)][static_cast<int>(ScalarType::NumOptions)] = {
    /*        u1  i1  i2  i4  i8  f2  f4  f8  c2  c4  c8  b1  q1  q2  q3  bf*/
    /* u1 */ {u1, i2, i2, i4, i8, f2, f4, f8, ud, c4, c8, u1, ud, ud, ud, ud},
    /* i1 */ {i2, i1, i2, i4, i8, f2, f4, f8, ud, c4, c8, i1, ud, ud, ud, ud},
    /* i2 */ {i2, i2, i2, i4, i8, f2, f4, f8, ud, c4, c8, i2, ud, ud, ud, ud},
    /* i4 */ {i4, i4, i4, i4, i8, f2, f4, f8, ud, c4, c8, i4, ud, ud, ud, ud},
    /* i8 */ {i8, i8, i8, i8, i8, f2, f4, f8, ud, c4, c8, i8, ud, ud, ud, ud},
    /* f2 */ {f2, f2, f2, f2, f2, f2, f4, f8, ud, c4, c8, f2, ud, ud, ud, ud},
    /* f4 */ {f4, f4, f4, f4, f4, f4, f4, f8, ud, c4, c8, f4, ud, ud, ud, ud},
    /* f8 */ {f8, f8, f8, f8, f8, f8, f8, f8, ud, c8, c8, f8, ud, ud, ud, ud},
    /* c2 */ {ud, ud, ud, ud, ud, ud, ud, ud, c2, c4, c8, ud, ud, ud, ud, ud},
    /* c4 */ {c4, c4, c4, c4, c4, c4, c4, c8, c4, c4, c8, ud, ud, ud, ud, ud},
    /* c8 */ {c8, c8, c8, c8, c8, c8, c8, c8, c8, c8, c8, ud, ud, ud, ud, ud},
    /* b1 */ {u1, i1, i2, i4, i8, f2, f4, f8, ud, ud, ud, b1, ud, ud, ud, ud},
    /* q1 */ {ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud},
    /* q2 */ {ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud},
    /* q3 */ {ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud},
    /* bf */ {ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, ud, bf},
  };
)
```

Figure 3.7. The part of code from the function *_promoteTypesLookup* which decides which data type to promote out of the given two data types. The table in the constant expression *_promoteTypesLookup* mimics the promotion rules from NumPy. The codes used for each data type are given in the code itself, like *u1* is *Byte* while *i1* represents *Char*.

Enabling type promotion for all the relevant unary ops (also called Unary Floating Universal Functions) does affect other parts like ONNX (Open Neural Network Exchange) and JIT (Just In Time).

Once the promotion rule is enabled, the next step is to make sure that ONNX Routine tests that integral arguments are properly casted to the default floating type. Not all the ops are

available in ONNX Opset version 9[14], and hence the ops which are not available are omitted from the list.

test_pytorch_onnx_onnxruntime.py

```
# Unary floating ufunc tests
# Tests that integral arguments are properly cast to float
unary_floating_ufuncs = (
    'sin', 'cos', 'tan',
    'asin', 'acos', 'atan',
    'sinh', 'cosh', 'tanh',
    'ceil', 'floor',
    'exp', 'log', 'log1p', 'log2',
    'sqrt', 'rsqrt',
    'erf',
    'sigmoid'
)

def setup_unary_floating_ufunc_tests_helper(cls, op_name):
    op = getattr(torch, op_name)

    def test_fn(self):
        class module(torch.nn.Module):
            def forward(self, x):
                return op(x)

        x = (torch.rand(2, 3, 4) * 10).to(torch.long)
        self.run_test(module(), (x,))
        self.run_test(torch.jit.trace(module(), (x,)), (x,))

    test_name = "test_" + op_name + "_ufunc"
    assert not hasattr(cls, test_name), "{0} already in {1}".format(test_name, cls.__name__)
    if op_name in ['erf', 'sinh', 'cosh']:
        setattr(cls, test_name, skipIfUnsupportedMinOpsetVersion(9)(test_fn))
    else:
        setattr(cls, test_name, test_fn)

def setup_unary_floating_ufunc_tests(cls):
    for op_name in unary_floating_ufuncs:
        setup_unary_floating_ufunc_tests_helper(cls, op_name)

setup_unary_floating_ufunc_tests(TestONNXRuntime)
```

Figure 3.8. Implementation of tests for ONNX Routine in PyTorch for unary floating universal functions type promotion. It's important to observe how *skipIfUnsupportedMinOpsetVersion(9)* is used for ops like *erf, sinh* and *cosh* since not all ops are present in all Opset Versions. These tests are to ensure the type promotion of integral tensors to floating type is done in ONNX.

The implementation of each unary floating universal function is also changed for ONNX to support type promotion. An example of this for *sqrt* is shown in the Figure 3.9.

torch/onnx/symbolic_helper.py

```
def _unary_ufunc_helper(g, self, op_name):
    if not _is_fp(self):
        type = scalar_type_to_pytorch_type.index(torch.get_default_dtype())
        self = g.op("Cast", self, to_i=scalar_type_to_onnx[type])

    return g.op(op_name, self)

def sqrt(g, self):
    // Before: return g.op("Sqrt", self)
    return sym_help._unary_ufunc_helper(g, self, "Sqrt")
```

Figure 3.9. Example implementation of a unary universal function *sqrt* in PyTorch's ONNX.

Once the ONNX Routine has been taken care of, the next step is to make sure that the JIT (Just In Time) compiler of PyTorch takes care of shape analysis for these unary floating universal functions. The reason this is required is, since the output of an op is updated because of type promotion, it needs to be made sure that the tests in JIT are updated as well. A new lambda function is created to ensure this, which is described in the Figure 3.10 below.

The target is to keep the same structure and logic like for other Unary Ops, but add the check and set the scalar type when appropriate. It's worth noting how a list of unary floating ops which promote using *PROMOTE* strategy from integer to floating type (default floating type) is passed to the lambda function. The list of the unary floating universal functions is given below;

acos, angle, tanh, asin, atan, ceil, cos, cosh, digamma, erf, erfc, erfinv, exp, expm1, log, log10, log1p, log2, floor, frac, lgamma, rsqrt, sigmoid, sin, sinh, sqrt, tan, tanh, trunc

The list is carefully discussed among the upstream and then added to the type promotion proposal. The idea is to have these ops supporting integral or boolean input tensors to floating point conversion implicitly without any output data type being passed.

torch/csrc/jit/passes/shape_analysis

```
// Requirements:
//   dims           : preserved
//   scalar type    : promoted to floating if integral/boolean, else preserved
//   device         : preserved
//   tensor inputs  : 1
//   tensor outputs : 1
// Additionally:
//   - First input should be the only tensor input
static const register_formula_for unary_floating_ufuncs{
  {
    "aten::acos(Tensor self) -> Tensor",
    "aten::angle(Tensor self) -> Tensor",
    "aten::tanh(Tensor self) -> Tensor",
    "aten::asin(Tensor self) -> Tensor",
    "aten::atan(Tensor self) -> Tensor",
    "aten::ceil(Tensor self) -> Tensor",
    "aten::cos(Tensor self) -> Tensor",
    "aten::cosh(Tensor self) -> Tensor",
    "aten::digamma(Tensor self) -> Tensor",
    "aten::erf(Tensor self) -> Tensor",
    "aten::erfc(Tensor self) -> Tensor",
    "aten::erfinv(Tensor self) -> Tensor",
    "aten::exp(Tensor self) -> Tensor",
    "aten::expm1(Tensor self) -> Tensor",
    "aten::log(Tensor self) -> Tensor",
    "aten::log10(Tensor self) -> Tensor",
    "aten::log1p(Tensor self) -> Tensor",
    "aten::log2(Tensor self) -> Tensor",
    "aten::floor(Tensor self) -> Tensor",
    "aten::frac(Tensor self) -> Tensor",
    "aten::lgamma(Tensor self) -> Tensor",
    "aten::rsqrt(Tensor self) -> Tensor",
    "aten::sigmoid(Tensor self) -> Tensor",
    "aten::sin(Tensor self) -> Tensor",
    "aten::sinh(Tensor self) -> Tensor",
    "aten::sqrt(Tensor self) -> Tensor",
    "aten::tan(Tensor self) -> Tensor",
    "aten::tanh(Tensor self) -> Tensor",
    "aten::trunc(Tensor self) -> Tensor",
  },
  [](Node* node) -> type_vec_t {
    if (auto type = node->input(0)->type()->cast<TensorType>()) {
      if (isIntegralType(*(type->scalarType()), /*includeBool=*/true)) {
        const auto default_type = at::typeMetaToScalarType(c10::get_default_dtype());
        auto ret = type->withScalarType(default_type);
        return type_vec_t{ret->dimensionedOnly()};
      }
      return type_vec_t{type->dimensionedOnly()};
    }
  return type_vec_t{};
  }};
  // ...
}
```

Figure 3.10. Implementation of the lambda function and the list for unary ops listed. It can be observed that all the unary floating universal functions are passed from a list to the lambda function to ensure shape analysis tests of JIT in PyTorch pass for completeness.

CHAPTER 5

Contributions

Preface

The chapter talks about the contributions made by the author to the PyTorch library, with the assistance of the team at NVIDIA and PyTorch. Since PyTorch is an open source library, maintained by the developers at Facebook, the future scope of work is not discussed here as the development culture is pretty dynamic. The chapter links to the pull requests made by the author to the PyTorch library which are relevant to type promotion and in general.

5.1 Contributions Made by the Author

The thesis has led to major contributions to PyTorch with the help of the author's mentors at NVIDIA (Christian Sarofeen and Michael Carilli) and a few developers at Facebook (Mike Ruberry and Brian Vaughn). The work was a hot topic to discuss amongst the developers in general and had been there to be worked at for a while. The author along with the team, helped initiate the type promotion pull request to the PyTorch source code. There were two relevant pull requests made to the PyTorch source code, for enabling type promotion (both implicit and explicit).

- Add floating type promotion support for (some of the) Unary Floating Universal Functions [15].
- Adding *dtype* argument to the Unary Functions for type promotion (testing on *expm1* function) [16].

Both of the pull requests listed are in the queue to be approved and imported, as there has been ongoing work on other relevant parts of PyTorch (like *TensorIterator*). Apart from the contributions to the type promotion of unary functions, the author has also contributed to PyTorch to fix bugs and add NumPy like functions to PyTorch.

- Adding *nanprod* to *ReduceOps* in PyTorch [17].
- Adding *nanmean* to *ReduceOps* in PyTorch [18].
- Adding *arccosh, arcsinh, arctanh* to *UnaryOps* in PyTorch [19].
- Remove duplicate complex type tests in PyTorch [20].
- Add *tanh_cuda* support for complex types [21].

- Solve issue, to make sure *torch.prod* (product function in PyTorch) works fine with FP16 input tensor and FP32 output tensor [22].

The author has added more than 1000 lines of code to the PyTorch library throughout this work. With more than 250 review comments in total, the work has been appreciated by the community.

References

[1] Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." Advances in neural information processing systems. 2019.
[2] PyTorch Alpha-1 Prerelease https://github.com/pytorch/pytorch/releases/tag/v0.1.1.
[3] PyTorch, the team https://github.com/pytorch/pytorch/releases/tag/v0.1.1.
[4] NVIDIA CUDA, High Performance Computing Platform for Deep Learning https://developer.nvidia.com/cuda-toolkit.
[5] NumPy, fundamental package for scientific computing in Python https://github.com/numpy/numpy.
[6] Documentation of PyTorch C++ Frontend API, Libtorch https://pytorch.org/cppdocs/.
[7] Frostig, Roy, Matthew James Johnson, and Chris Leary. "Compiling machine learning programs via high-level tracing." Systems for Machine Learning (2018).
[8] Composable transformations of Python+NumPy programs: differentiate, vectorize, JIT to GPU/TPU, and more https://github.com/google/jax.
[9] Stallman, Richard, Roland Pesch, and Stan Shebs. "Debugging with GDB." Free Software Foundation 675 (1988).
[10] Contributing to PyTorch and building from source: https://github.com/pytorch/pytorch/blob/master/CONTRIBUTING.md#developing-pytorch.
[11] Gough, Brian J., and Richard Stallman. An Introduction to GCC. Network Theory Limited, 2004.
[12] TensorIterator API of PyTorch, source code and header file: https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/TensorIterator.cpp and https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/TensorIterator.h.
[13] Implementation of Unary Ops (Functions) in PyTorch: https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/UnaryOps.cpp.
[14] ONNX Operators https://github.com/onnx/onnx/blob/master/docs/Operators.md.
[15] Pull Request to add floating type promotion support for some of the unary floating universal functions, authored by Kushashwa Ravi Shrimali on 14th February 2020: https://github.com/pytorch/pytorch/pull/33322.
[16] Pull Request to add dtype argument to the Unary Functions for type promotion (initial testing on expm1 function), authored by Kushashwa Ravi Shrimali on 7th February 2020: https://github.com/pytorch/pytorch/pull/33063.
[17] Pull Request to add nanprod function to ReduceOps (Reduction Operators) in PyTorch, authored by Kushashwa Ravi Shrimali on 18th May 2020 https://github.com/pytorch/pytorch/pull/38634
[18] Pull Request to add nanmean function to ReduceOps (Reduction Operators) in PyTorch, authored by Kushashwa Ravi Shrimali on 18th May 2020:

https://github.com/pytorch/pytorch/pull/38632.
[19] Pull Request to add arccosh, arcsinh, arctanh to UnaryOps (Unary Operators) in PyTorch, authored by Kushashwa Ravi Shrimali on 13th May 2020: https://github.com/pytorch/pytorch/pull/38388.
[20] Pull Request to remove duplicate complex type tests in PyTorch, authored by Kushashwa Ravi Shrimali on 19th May 2020: https://github.com/pytorch/pytorch/pull/38713.
[21] Pull Request to add support for complex types for tanh function on CUDA devices, authored by Kushashwa Ravi Shrimali on 14th May 2020: https://github.com/pytorch/pytorch/pull/38486.
[22] Pull Request to solve issue for torch.prod functioning on floating point 16 input tensor and floating point 32 output tensor, authored by Kushashwa Ravi Shrimali on 30th January 2020: https://github.com/pytorch/pytorch/pull/32831.