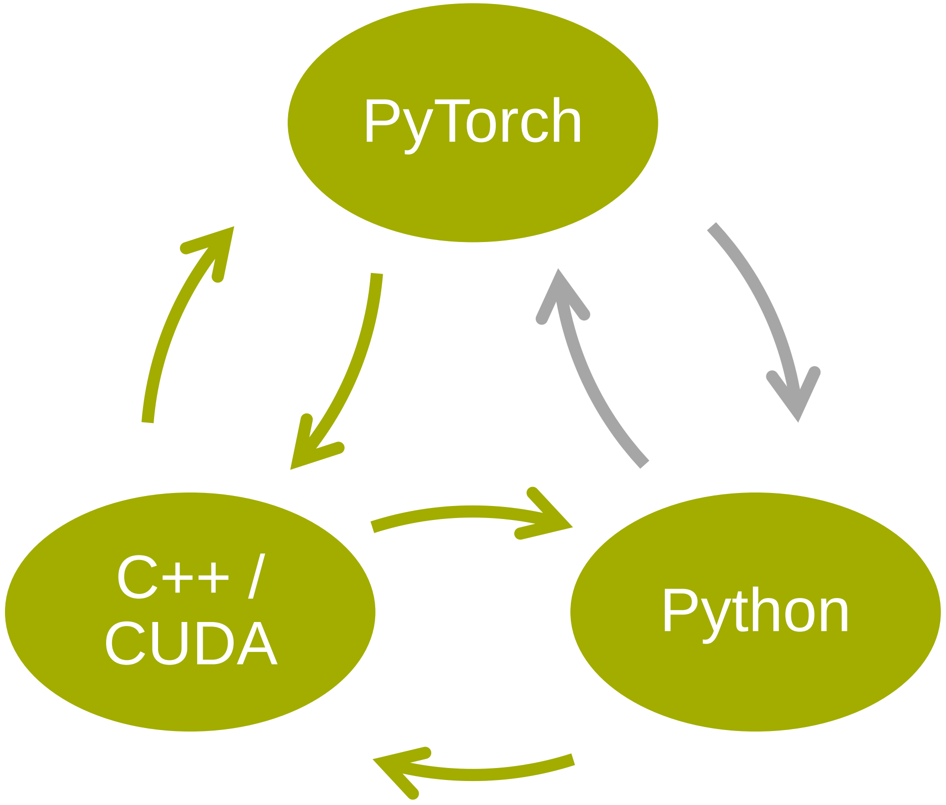
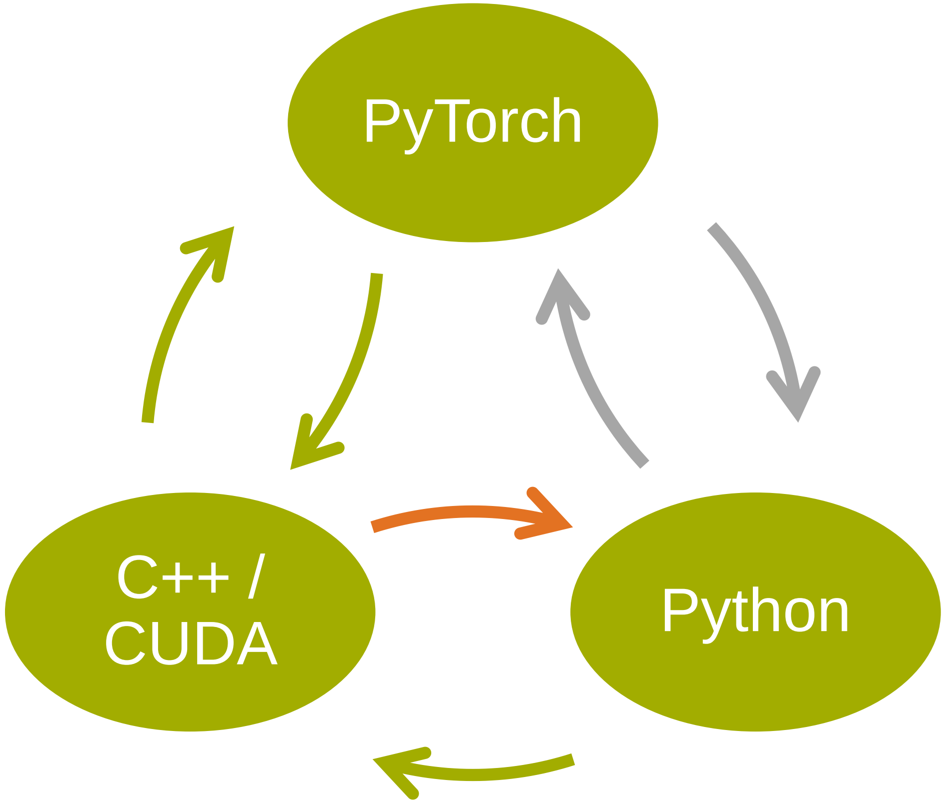
Pytorch – C++ – CUDA Interoperability



Sebastian Weiß

1. Calling C++ from Python



1. Calling C++ from Python

With `ctypes`

C++:

```
extern "C" __declspec(dllexport)
int foo(const char* str, float bar)
{ return 1; }
```

➔ Compile as .dll / .so

Python:

```
import ctypes
lib = ctypes.cdll.LoadLibrary("Lib.dll")
lib.foo.artypes = [ctypes.c_char_p, ctypes.c_float]
lib.foo.restype = ctypes.c_int
lib.foo("str", 42.0) # call it
```

+ non-intrusive

- Duplicate type specification

With *pybind11* (https://github.com/pybind/pybind11)

C++:

```
#include <pybind11/pybind11.h>
int foo(const char* str, float bar);
PYBIND11_MODULE(MyModule, m) {
    m.def("foo", &foo);
}
```

➔ Compile as .dll / .so and rename to .pyd

Python:

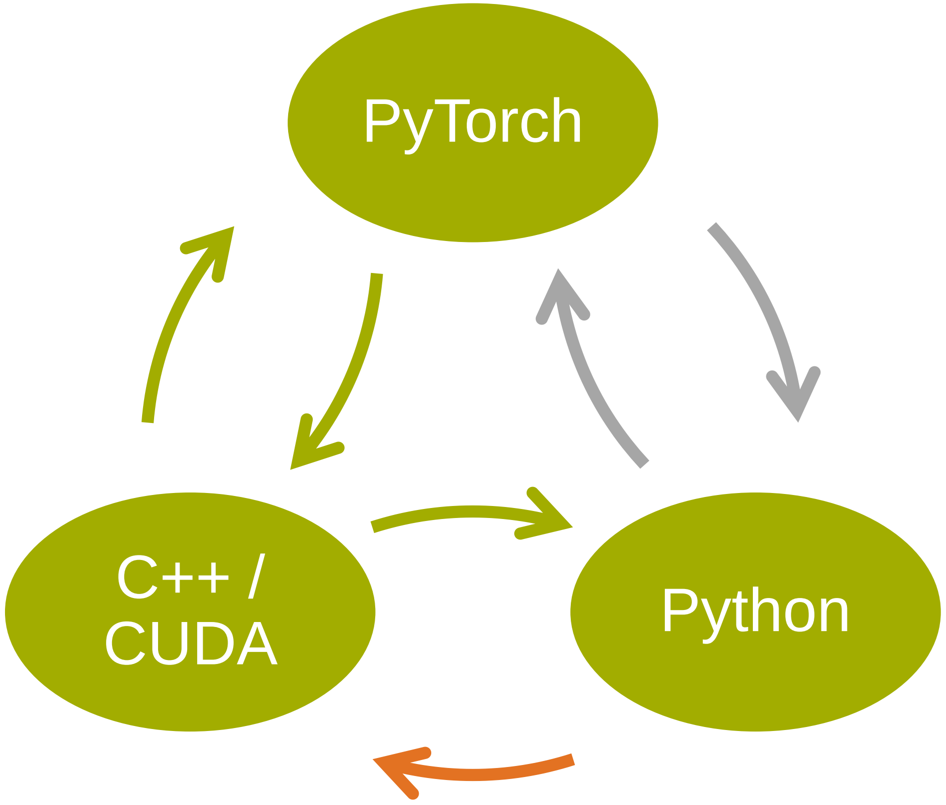
```
import MyModule as mm
ret = mm.foo("str", 42)
```

+ Supports functions, classes, enums, …

+ Supports doc-strings and optional params

- Requires wrapper-code in C++

2. Calling Python from C++



2. Calling Python from C++

With the Python C-API

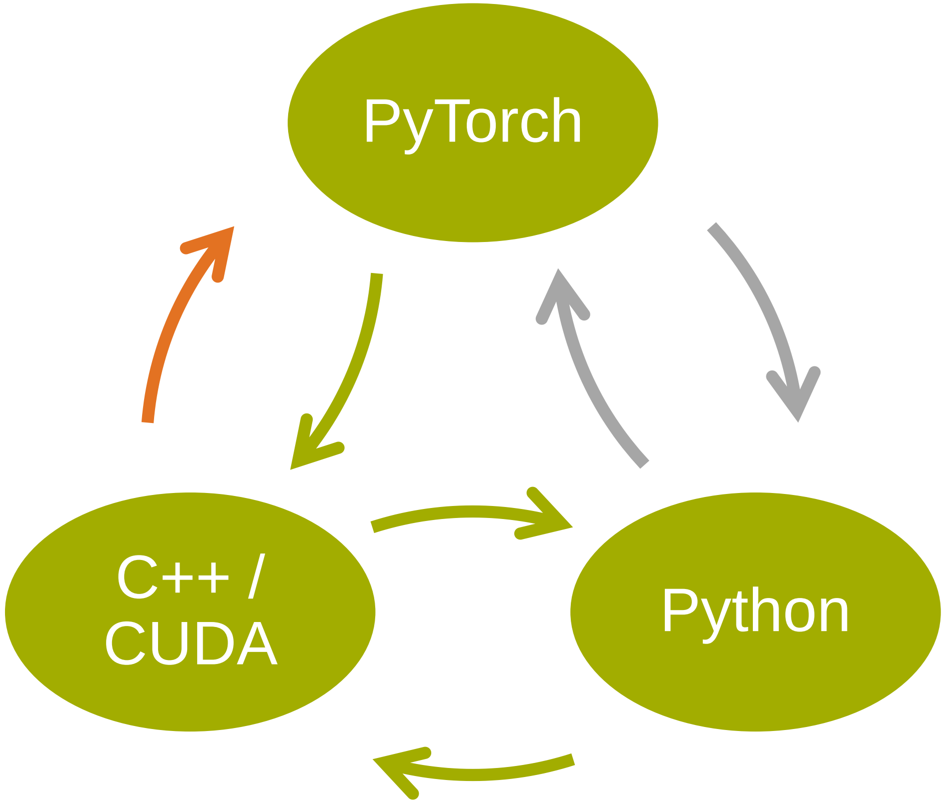
DON´T!!

With pybind11

```
py::object os = py::module::import("os");
py::object makedirs = os.attr("makedirs");
makedirs("/tmp/path/to/somewhere");
```

➔ Read the docs for more infos

3. Calling C++ from PyTorch



3. Calling C++ from PyTorch

- Use CMAKE to link against PyTorch
- Includes and Libraries are shipped with the Python distribution
- Stand-alone TorchLib available

```
find_package(Torch REQUIRED)
target_include_directories(MyTarget
      PUBLIC ${TORCH_INCLUDE_DIR})
target_link_libraries(MyTarget ${TORCH_LIBRARIES})
```

We want to bind the following operation to PyTorch:

```
torch::Tensor custom_add(torch::Tensor a, torch::Tensor b)
{
    return a + b + 1;
}
```

3. Calling C++ from PyTorch

With pybind11

C++:

```
PYBIND11_MODULE(MyModule, m) {
    m.def("custom_add", &custom_add);
}
```

➔ Compile as .dll / .so and rename to .pyd

Python:

```
import MyModule as mm
custom_add.foo(torch.rand(2,4), …)
```

+ Simple
- Not visible in the computation graph!
- Does not work with TorchScript

With TorchScript

C++:

```
#include <torch/script.h>
static auto registry = torch::jit::RegisterOperators()
.op("mymodule::add", & custom_add);
```

➔ Compile as .dll / .so

Python:

```
import torch
torch.ops.load_library("./MyModule.dll")
torch.ops.mymodule.add(
   torch.rand(2,4), torch.rand(2,4))
```

+ Supports TorchScript
- Limited parameter types
  int64, double, std::string, bool, Tensor of any type

4. Writing CUDA-Ops

Example: Operation that outputs a 2D float tensor

Includes:

```
#include <cuMat/src/Errors.h>
#include <cuMat/src/Context.h>
#include <torch/types.h>
#include <ATen/cuda/CUDAContext.h>
```

Kernel:

```
__global__ void FillIndexKernel(
  dim3 virtual_size,
  torch::PackedTensorAccessor<float, 2 /*Dim*/>
        output)
{
  CUMAT_KERNEL_2D_LOOP(x, y, virtual_size)
    output[y][x] = x + y; //some operation
  CUMAT_KERNEL_2D_LOOP_END
}
```

4. Writing CUDA-Ops

Example: Operation that outputs a 2D float tensor

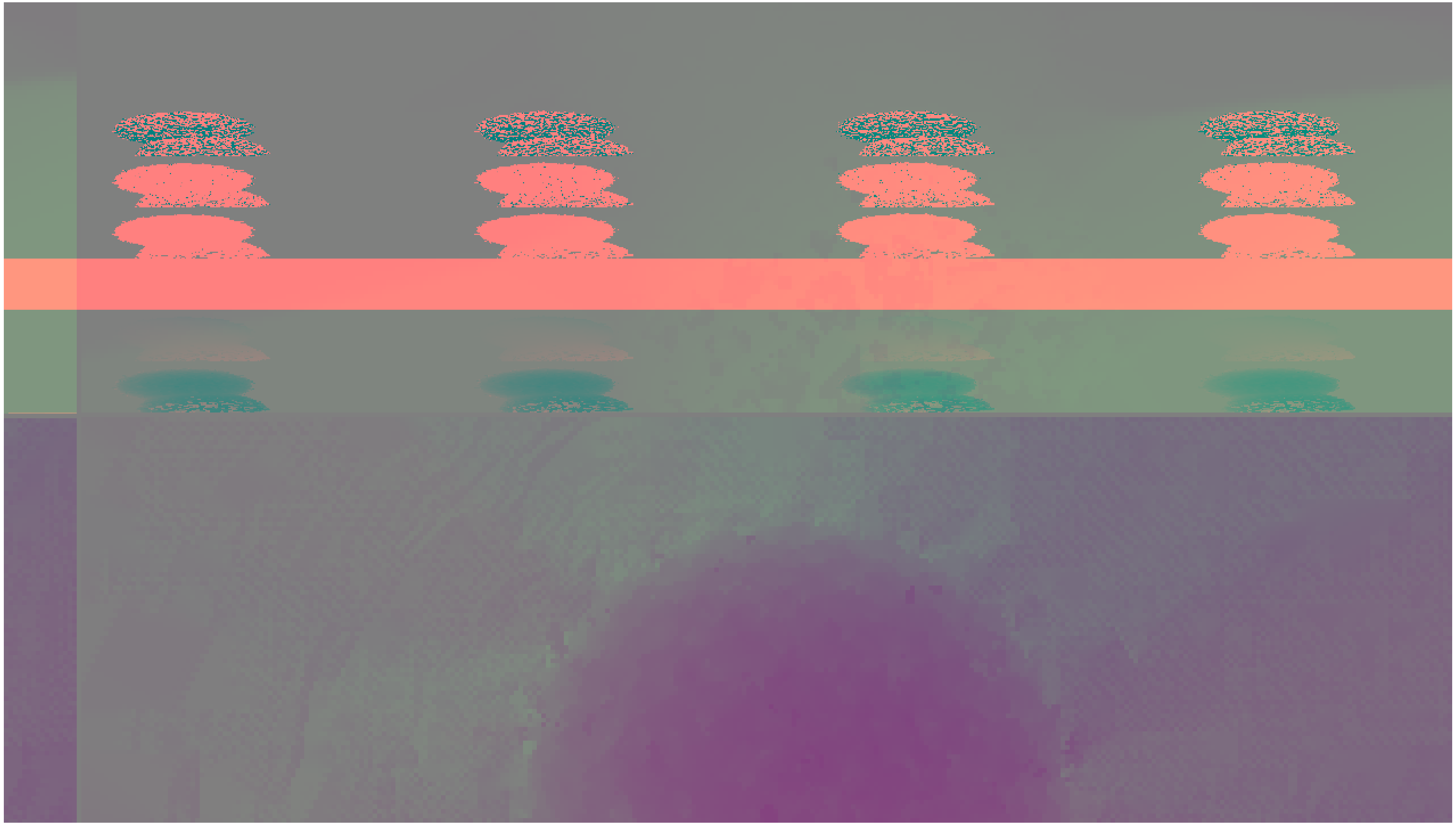
Launch:

```
torch::Tensor FillIndex(
    int64_t width, int64_t height)
{
//allocate output, note the height*width convention
torch::Tensor out = torch::empty(
  /*size*/ { height, width },
  /*options*/ torch::device(at::kCUDA)
              .dtype(at::kFloat));

// CUDA stream for synchronization with PyTorch
cudaStream_t stream =
  at::cuda::getCurrentCUDAStream();

// use cuMat to compute launch bounds
cuMat::Context& ctx = cuMat::Context::current();
cuMat::KernelLaunchConfig cfg =
  ctx.createLaunchConfig2D(
     width, height, FillIndexKernel);

//launch the kernel
FillIndexKernel
 <<< cfg.block_count, cfg.thread_per_block, 0, stream >>>
 (cfg.virtual_size, out.packed_accessor<float, 2>());
CUMAT_CHECK_ERROR();

return out;
}
```

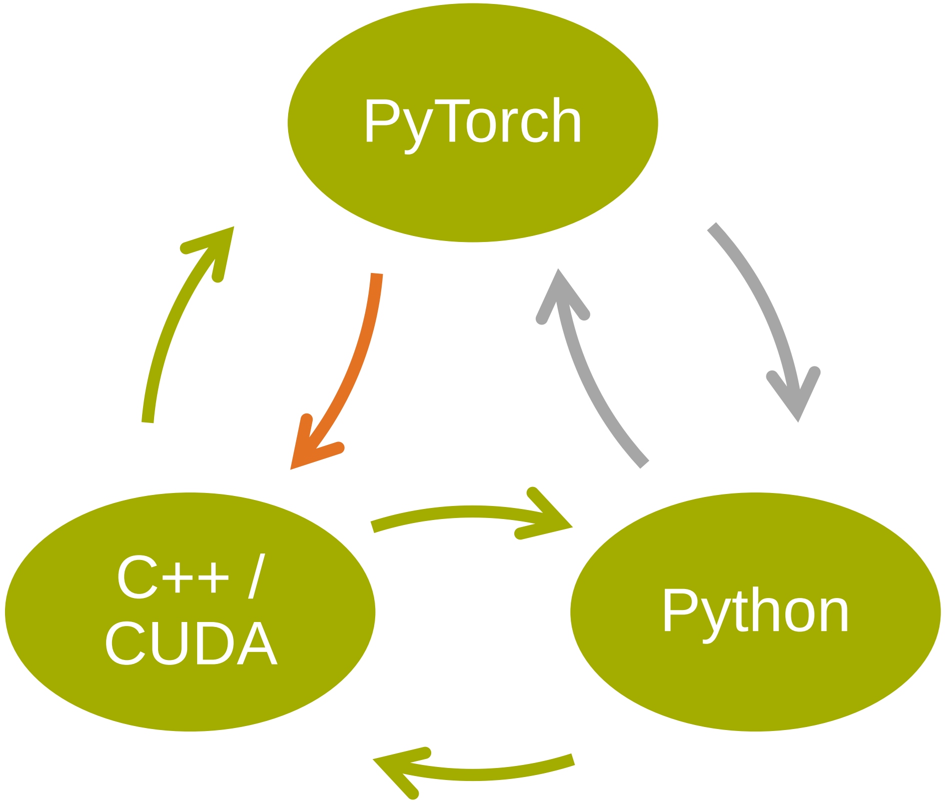
Registration as usual

4. Writing CUDA-Ops

Synchronization with the PyTorch stream is important!



5. Calling PyTorch from C++



5. Calling PyTorch from C++

Given the following model

```
class MyModule(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, a, b):
        device = torch.device('cuda')
        a = a.to(device)
        b = b.to(device)
        return a + b
```

Convert it to a script

```
# 1)
scripted_module = torch.jit.script(MyModule())
# 2)
scripted_module = torch.jit.trace(
   MyModule(), torch.rand(5, 3), torch.rand(5,3))

torch.jit.save(scripted_module, "test.pt")
```

Script `# 1)`

+ Allows dynamic control flow

- Not all ops are supported

Trace `# 2)`

+ All ops are supported

- No dynamic control flow
  (constant branching allowed)

5. Calling PyTorch from C++

test.pt:

```
...
def forward(self,
    a: Tensor,
    b: Tensor) -> Tensor:
  device = torch.device("cuda")
  a0 = torch.to(a, device, None, False, False)
  b0 = torch.to(b, device, None, False, False)
  return torch.add(a0, b0, alpha=1)
...
```

5. Calling PyTorch from C++

EnhanceNet.pt (trace)

```
def forward(self,
    input: Tensor) -> Tuple[Tensor, Tensor]:
  _0 = getattr(self.preblock, "0")
  weight = _0.weight
  _1 = _0.bias
  _2 = self.blocks
  _3 = getattr(_2, "0")
  _4 = getattr(_3, "0")
  weight0 = _4.weight
  _5 = _4.bias
  _6 = getattr(_3, "2")
  weight1 = _6.weight
  _7 = _6.bias
  _8 = getattr(_2, "1")
  _9 = getattr(_8, "0")
  weight2 = _9.weight
  _10 = _9.bias
  _11 = getattr(_8, "2")
  weight3 = _11.weight
  _12 = _11.bias
  _13 = getattr(_2, "2")
  _14 = getattr(_13, "0")
…
```

```
…
 input0 = torch._convolution(input, weight, _1, [1, 1], [1,
1], [1, 1], False, [0, 0], 1, False, False, True)
  input1 = torch.relu(input0)
  input2 = torch._convolution(input1, weight0, _5, [1, 1],
[1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  input3 = torch.relu(input2)
  _60 = torch._convolution(input3, weight1, _7, [1, 1], [1,
1], [1, 1], False, [0, 0], 1, False, False, True)
  input4 = torch.add(input1, _60, alpha=1)
  input5 = torch._convolution(input4, weight2, _10, [1, 1],
[1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  input6 = torch.relu(input5)
…
  input32 = torch.upsample_bilinear2d(input31, _76, False)
  input33 = torch._convolution(input32, weight20, _55, [1,
1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  input34 = torch.relu(input33)
  input35 = torch._convolution(input34, weight21, _57, [1,
1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  input36 = torch.relu(input35)
  outputs = torch._convolution(input36, weight22, _59, [1,
1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
…
```

5. Calling PyTorch from C++

```cpp
#include <torch/script.h>

//create example tensors
torch::Tensor t1 = torch::randn({ 4, 8 },
  torch::dtype(at::kFloat).device(c10::kCUDA));
torch::Tensor t2 = torch::randn({ 4, 8 },
  torch::dtype(at::kFloat).device(c10::kCUDA));

//load scripted module
torch::jit::script::Module module =
  torch::jit::load("test.pt");

//run module
torch::Tensor out =
  module.forward({t1, t2}).toTensor();
```

6. Lessons Learned

1. Mind the datatypes

   Numpy, Python „float“: 64-bit float

   PyTorch, C++ „float“: 32-bit float

   → always use `np.float32` in the training code

2. Tensors (`torch::Tensor`) by default store intermediate results for backprop

   ➔ Disable gradients during inference to optimize memory

- Python:

```python
with torch.no_grad():
    …
```

- C++:

```cpp
at::GradMode::set_enabled(false);
```

6. Lessons Learned

3. C++/CUDA-Extensions: additional constraints on the function types in `torch::jit::RegisterOperators().op(…)`
   - Function must return a value, „void“ is not allowed
   - Only primite types or „`const Tensor&`“ allowed, mutable „`Tensor&`“ invalid