SambaNova SYSTEMS

Modeling: PyTorch to SambaFlow

July 11, 2023

How Does It Work?

1. Import your model from PyTorch
2. Run **`samba.from_torch_model_(...)`** to convert model parameters to SambaTensors
    a. Convert input Torch Tensors with **`samba.from_torch_tensor(...)`**
3. Run **`samba.utils.common.common_app_driver(...)`** to compile the model
    a. Sometimes requires adaptations for compatibility (more details coming up)
4. Start running - benchmark or train

What are SambaTensors?

- Wrappers around Torch Tensors, with special SN capabilities
- Each one gets a unique **`name`** for interfacing with the chip
- Specify **`batch_dim`** for optimization
- Methods to transfer to/from RDU
- Can be treated like normal Torch Tensor,

  e.g., `tensor3 = tensor1 + tensor2`

  or `tensor2 = tensor1.reshape(-1, 5)`

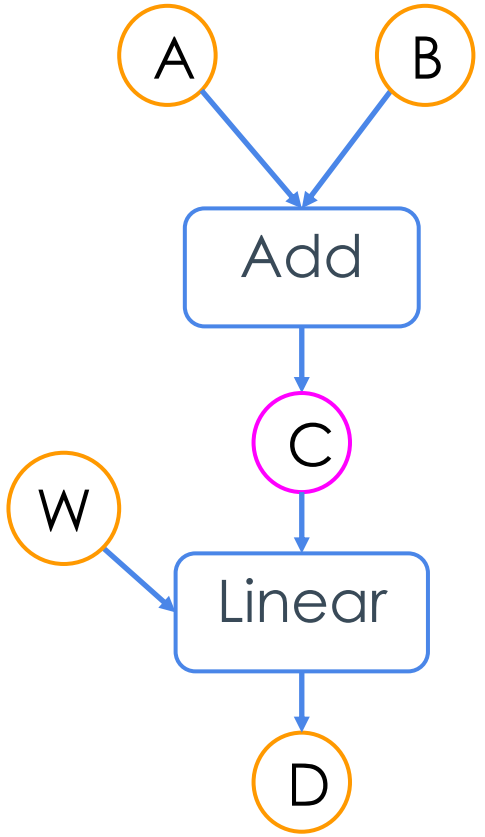
SambaTensor

.data (torch tensor)
.batch_dim (int)
.dtype (torch dtype)
.sn_name (string)
.sn_grad (gradient on chip)

.cpu() (transfer to CPU)
.rdu() (transfer to RDU)

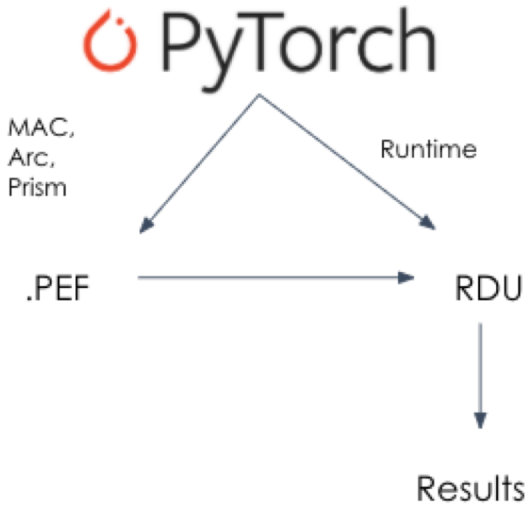
What is tracing?

- Tracing walks through the model with dummy tensors to form the graph
- This is carried out automatically during compilation or it can be done manually by using **`trace_graph()`** before running on RDU
- Beneficial because the RDU+compiler have knowledge of the **full graph** to optimize, not just individual components



How do we trace your graph?

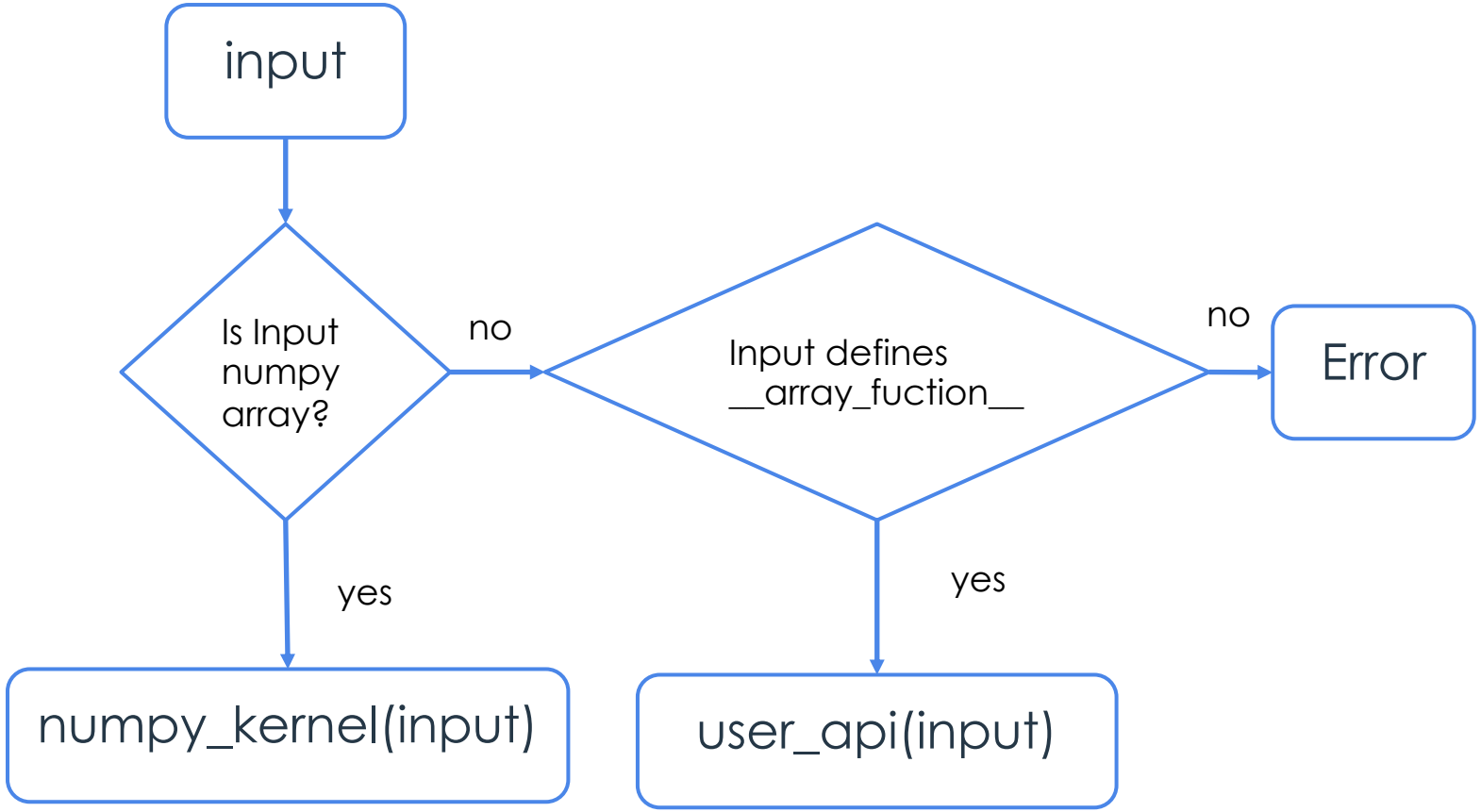
- Model definition is done in PyTorch
- The model's **forward pass** will determine the graph that is generated
  - Note that there is no current support for control flow within model
- For compiling: **samba.utils.common.common_app_driver**
- For training: **samba.utils.utils.trace_graph**



Numpy/PyTorch Functional Overrides

- Numpy style API dispatch:

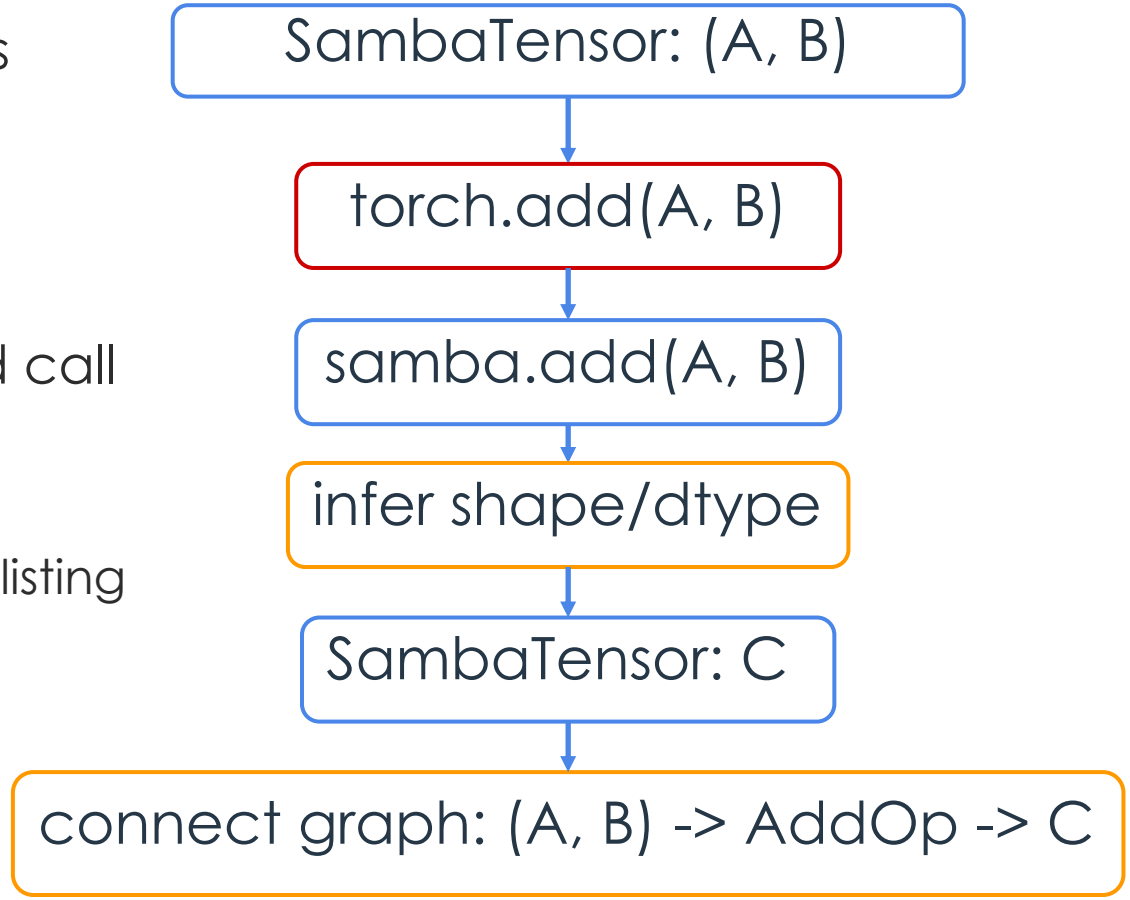
```
a = torch.ones(2, 4)
b = torch.zeros(2, 4)

np.add(a, b)

tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.]])
```



https://pytorch.org/docs/stable/notes/extending.html#extending-torch-with-a-tensor-like-type

Functional Overrides with SambaTensor

- SambaFlow understands and supports similar functional overrides
- If you pass SambaTensors to a Torch function, SambaFlow will override and call the equivalent SambaFlow method
  - Check the SambaFlow API docs for a listing of methods



Model Parameters

- Operations with weights have internal parameters

- Convert Torch parameters to Samba parameters in-place by traversing model by using **samba.from_torch_model_(model)**

Takeaways So Far

- Start with a PyTorch model
- Convert parameters to Samba format using **samba.from_torch_model_** and Torch Tensors to SambaTensors using **samba.from_torch_tensor**
- Pass in **SambaTensors** during tracing
- Use **samba.utils.common.common_app_driver** (to compile) or **samba.utils.utils.trace_graph** to trace (during runtime)

Sample Code (App Code)

- Running is 2 Steps, **compile** then **run**
- Need to pass **model**, **inputs**, and **optimizer** to compile/trace
- Write your own training loop!

```
#Input definition
image = samba.randn(args.batch_size, args.num_features, name='image', batch_dim=0)
label = samba.randint(args.num_classes, (args.batch_size, ), name='label', batch_dim=0)
inputs = (image, label)

# Model definition
model = ResFFNLogReg(args.num_features, args.hidden_size, args.num_classes)
samba.from_torch_model_(model)

# Optimizer definition
optim = sambaflow.samba.optim.SGD(model.parameters(),
                                  lr=args.lr,
                                  momentum=args.momentum,
                                  weight_decay=args.weight_decay)

# Different run modes
# The common_app_driver() handles model compilation and various other tasks, e.g.,
# measure-performance. Running, or training, a model must be explicitly carried out
if args.command == "run":
    # Trace the graph
    utils.trace_graph(model, inputs, optim, pef=args.pef, mapping=args.mapping)
    # Within the user defined train function, call: samba.session.run
    train(args, model)
else:
    common_app_driver(args=args,
                      model=model,
                      inputs=inputs,
                      optim=optim,
                      name=model.__class__.__name__,
                      init_output_grads=not args.inference,
                      app_dir=utils.get_file_dir(__file__))
```

SambaNova SYSTEMS

Sample Code (Model)

- Model code all in PyTorch
- A few restrictions
  - Modules with parameters are defined before forward
  - The return type of forward should be simple (tensor, tuple/list of tensors)

```
class ResFFNLogReg(nn.Module):
    """Feed Forward Network with two different activation functions and a residual connection"""
    def __init__(self, num_features: int, hidden_size: int, num_classes: int) -> None:
        super().__init__
        self.gemm1 = nn.Linear(num_features, hidden_size, bias=True)
        self.gemm2 = nn.Linear(hidden_size, hidden_size, bias=True)
        self.gemm3 = nn.Linear(hidden_size, num_classes, bias=True)

        self.norm1 = nn.LayerNorm(hidden_size)
        self.norm2 = nn.LayerNorm(hidden_size)

        self.tanh1 = nn.Tanh()
        self.sigmoid1 = nn.Sigmoid()

        self.criterion = nn.CrossEntropyLoss()

        self.apply(basic_weight_init)

    def forward(self, inputs: torch.Tensor, targets: torch.Tensor) -> Tuple[torch.Tensor]:
        out = self.gemm1(inputs)
        out = self.norm1(out)
        out = self.tanh1(out)
        residual = out
        out = self.gemm2(out)
        out = self.norm2(out)
        out = out + residual
        out = self.sigmoid1(out)
        out = self.gemm3(out)
        loss = self.criterion(out, targets)
        return loss, out
```

SambaNova SYSTEMS

Sample Code (Tracing)

- If a **SambaTensor** goes through a **Torch Function/Module** it will be traced
- This includes many functions in torch.nn.functional
  - Built-in torch modules
  - Torch functions/activations
  - Tensor functions

```
module = torch.nn.Conv2d(kernel_size=(3, 3),
                         in_channels=3,
                         out_channels=64,
                         padding=1,
                         bias=True)

# batch_size, in_channels, width, height
x = samba.randn(100, 3, 64, 64, name='input', batch_dim=0)

# If samba.session.tracing is set, this will trace the convolution operation
y = module(x)
```

Module tracing

```
import torch.nn.functional as F
x = samba.rand(10, 20, name='input', batch_dim=0)

# Similarly, during tracing these will get traced
y = F.relu(x)
z = x.split(10, dim=1)
```

Function tracing

SambaNova SYSTEMS

Sample Code (Run, pt. 1)

- Run the model with **samba.session.run**
  - Provide all **input_tensors**
  - Provide traced **output_tensors**
- Running specific sections
  - **section_ids**
  - **section_types**
- Sync parameter values between Host/RDU

```
outputs = samba.session.run(input_tensors=inputs,
                            output_tensors=traced_outputs,
                            section_types= ['fwd', 'bckwd', 'opt'])

# Sync the parameter values on rdu back to host
samba.session.to_cpu(model)

# Retrieve individual gradients
linear_grad = model.linear1.weight.sn_grad
```

Running all sections

```
for inputs in dataloader:
    # Running only fwd here for inference
    loss, out = samba.session.run(input_tensors = inputs,
                                  output_tensors = traced_outputs,
                                  section_types = ['fwd'])

    # Can do anything you like with the outputs
    process_inference_results(loss, out)
```

Sample inference loop

SambaNova SYSTEMS

Sample Code (Run, pt. 2)

- Partial model on RDU
  - Backpropagate gradients on host
- Define intermediate tensors at cut as ‘outputs’
- Do a second **samba.session.run** with **grad_of_outputs**
- This is typically done when using a custom loss function

```
traced_outputs = utils.trace_graph(model, inputs, optim)
partial_output = samba.session.run(input_tensors=inputs,
                                   output_tensors=traced_outputs,
                                   section_types=['fwd'])[0]

# Make PyTorch return the grads
partial_output = samba.to_torch(partial_output)
partial_output.requires_grad = True

loss = criterion(partial_output, targets)
loss.backward()

samba.session.run(input_tensors=inputs,
                  output_tensors=traced_outputs,
                  grad_of_outputs=(samba.from_torch_tensor(partial_output.grad),),
                  section_types=['bckwd', 'opt'])
```

Sample Code (Run, pt. 3)

- **samba.session.run** needs certain parameters at runtime, fed in as **argins**
  - Used for things like LR schedules, etc.
- Examples:
  - Learning rate（**lr**）
  - Weight decay（**weight_decay**）
  - SGD momentum（**momentum**）
  - Dropout rate（**p**）

```
base_lr = 0.01

for batch_num, input_batch in enumerate(dataloader):
    # Generate a new lr per iteration
    new_lr = lr_schedule(base_lr, batch_num)
    hyperparam_dict = {'lr': new_lr, 'weight_decay': weight_decay}

    samba.session.run(input_tensors=input_batch,
                      output_tensors=traced_outputs,
                      hyperparam_dict=hyperparam_dict)
```

Sample Code (Measuring Performance)

- Can measure performance of full model or each section
- Call respective function with sample inputs
- Use the **run_graph_only** option
- Will give you the throughput/latency for your model

```
# Trace the model
utils.trace_graph(model, inputs, optim, pef=args.pef)
utils.measure_performance(model, inputs,
                          batch_size=args.batch_size,
                          inference=args.inference,
                          run_graph_only=args.run_graph_only,
                          n_iterations=args.num_iterations)

utils.measure_sections(model, inputs,
                       batch_size=args.batch_size,
                       num_sections=num_sections,
                       n_iterations=args.num_iterations)
```

SambaNova SYSTEMS

Various args and run modes

- Args used internally expressed as command line args
- Some important args for compile/run:
  - `command`
  - `--inference`
  - `--batch-size/-b, --microbatch-size/-mb`
  - `--pef/-p`

An example compile command:

```
python <app>.py compile -b=64 -mb=4
--inference -p <app.pef>
```



```
args = parse_app_args(argv)
args.command == “compile”
args.batch_size == 64
args.microbatch_size == 4
args.inference == True
```

An example run command:

```
python <app>.py run -b=64 -mb=4
--inference -p <app.pef>
```



```
args = parse_app_args(argv)
args.command == “run”
args.batch_size == 64
args.microbatch_size == 4
args.inference == True
```

SambaNova SYSTEMS

Best Practices, dos and don’ts

- Inputs should be well-defined including internal tensors
  - Need to allow the compiler to see every symbol
- Try to contain operations to 1 continuous graph
- Avoid control flow within model
- Avoid synchronizing between CPU/RDU too often
- Stick with PyTorch, converting models and tensors as needed

SambaLoader

- The SambaLoader is wrapper around the PyTorch DataLoader
- It helps to improve overall performance by better parallelizing load ops and graph ops
- As a bonus, it returns an iterator over SambaTensors so you don't need to explicitly do that conversion!

Basic Method Structure:

```
SambaLoader(torch_loader: Iterable[Iterable[torch.Tensor]], names: List[str])

Where torch_loader is an Iterable and names is a list of strings to be given to the input SambaTensors
```

Example:

```
from torch.utils.data import DataLoader
from sambaflow.samba.sambaloader import SambaLoader

data_loader  = DataLoader(dataset, batch_size=args.bs,...)
samba_loader = SambaLoader(data_loader, [“sample”, “label”])

for X_val, Y_val in samba_loader:
    samba.session.run(input_tensors=[X_val, Y_val],...)
```

SambaLoader using Pinned Memory

- As in PyTorch, it prevents pinned pages from being paged out of memory
- It also parallelizes the transfer of Torch tensors to pinned memory with the graph run
- Needs to be enabled as part of a Samba Session

Example:

```
from torch.utils.data import DataLoader
from sambaflow.samba.sambaloader import SambaLoader

data_loader  = DataLoader(dataset, batch_size=args.bs,...)
samba_loader = SambaLoader(data_loader, [“sample”, “label”])

sambaflow.samba.session.enable_pinned_memory()

for X_val, Y_val in samba_loader:
    samba.session.run(input_tensors=[X_val, Y_val],...)
```