NVIDIA

TRTORCH

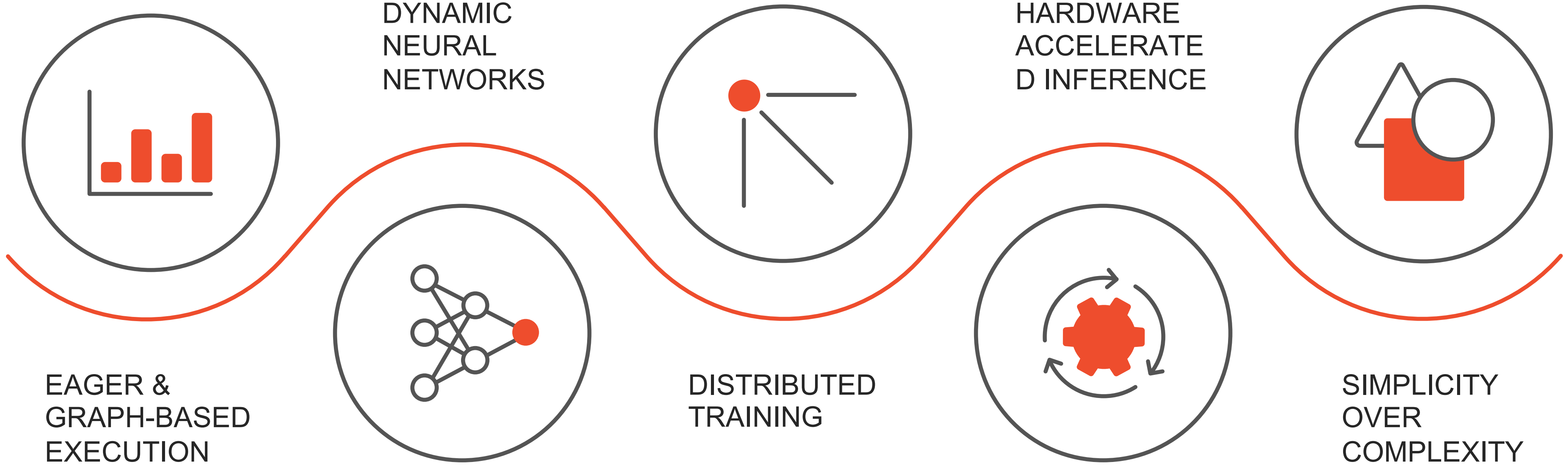
ACCELERATING PYTORCH INFERENCE USING TENSORRT

NAREN DASAN (NVIDIA) - AUTOMOTIVE DEEP LEARNING SOLUTIONS ARCHITECT
CHRIS GOTTBRATH (FACEBOOK) - TECHNICAL PROGRAM MANAGER - PYTORCH
JOSH PARK (NVIDIA) - AUTOMOTIVE SOLUTIONS ARCHITECT MANAGER

Agenda

- Intro
- PyTorch Torchscript
- TensorRT
- TRTorch

WHAT IS PYTORCH?



EAGER &
GRAPH-BASED
EXECUTION

DYNAMIC
NEURAL
NETWORKS

DISTRIBUTED
TRAINING

HARDWARE
ACCELERATE
D INFERENCE

SIMPLICITY
OVER
COMPLEXITY

PYTORCH

RESEARCH
PROTOTYPING

+

PRODUCTION
DEPLOYMENT

TORCHSCRIPT

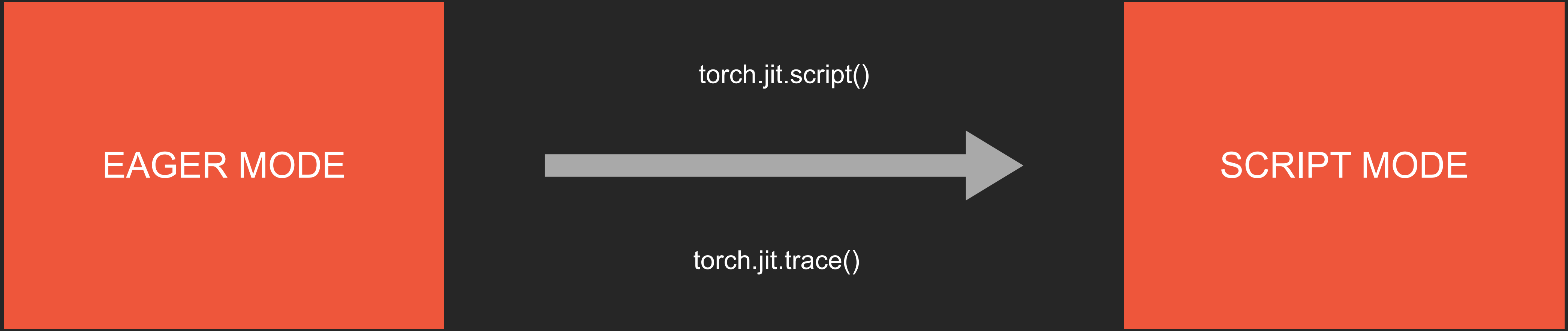
Models are Python TorchScript *programs,* an optimizable subset of Python

+ Same “models are programs” idea
+ Production deployment
+ No Python dependency
+ Compilation for performance optimization

```python
class RNN(nn.Module):
 def __init__(self, W_h, U_h, W_y, b_h, b_y):
  super(RNN, self).__init__()
  self.W_h = nn.Parameter(W_h)
  self.U_h = nn.Parameter(U_h)
  self.W_y = nn.Parameter(W_y)
  self.b_h = nn.Parameter(b_h)
  self.b_y = nn.Parameter(b_y)
 def forward(self, x, h):
  y = []
  for t in range(x.size(0)):
   h = torch.tanh(x[t] @ self.W_h + h @ self.U_h + self.b_h)
   y += [torch.tanh(h @ self.W_y + self.b_y)]
   if t % 10 == 0:
    print("stats: ", h.mean(), h.var())
  return torch.stack(y), h


# one annotation!
script_rnn = torch.jit.script(RNN(W_h, U_h, W_y, b_h, b_y))
```

TOOLS TO TRANSITION FROM EAGER TO SCRIPT



For prototyping, training, experimenting

For production deployment

EAGER TO SCRIPT MODE WITH torch.jit.trace()

Take an existing eager model, and provide example inputs.

The tracer runs the function, recording the tensor operations performed.

We turn the recording into a TorchScript module.

— Can reuse existing eager model code
— ⚠️ Control-flow and data structures are ignored

```python
import torch
import torchvision

def foo(x, y):
  return 2 * x + y

# trace a model by providing example inputs
traced_foo = torch.jit.trace(foo,
                             (torch.rand(3), torch.rand(3)))

traced_resnet = torch.jit.trace(torchvision.models.resnet18(),
                                torch.rand(1, 3, 224, 224))
```

EAGER TO SCRIPT MODE WITH torch.jit.script()

Write model directly in TorchScript, a high-performance subset of Python.

Pass instance of your model to torch.jit.script()

— Control-flow is preserved
— print statements can be used for debugging
— Remove the script() call to debug as a standard
PyTorch module

```
class RNN(torch.nn.Module):
  def __init__(self, W_h, U_h, W_y, b_h, b_y):
    super(RNN, self).__init__()
    self.W_h = nn.Parameter(W_h)
    self.U_h = nn.Parameter(U_h)
    self.W_y = nn.Parameter(W_y)
    self.b_h = nn.Parameter(b_h)
    self.b_y = nn.Parameter(b_y)

  def forward(self, x, h):
    y = []
    for t in range(x.size(0)):
      h = torch.tanh(x[t] @ self.W_h + h @ self.U_h + self.b_h)
      y += [torch.tanh(h @ self.W_y + self.b_y)]
      if t % 10 == 0:
        print("stats: ", h.mean(), h.var())
    return torch.stack(y), h

rnn = RNN(torch.randn(3, 4), torch.randn(4, 4), torch.randn(4, 4),
torch.randn(4), torch.randn(4))
scripted = torch.jit.script(rnn)
```

EXPORTING A MODEL TO PRODUCTION

TorchScript models can be saved a model archive and loaded to run in PyTorch's just-in-time (JIT) compiler instead of the CPython interpreter.

C++ Tensor APIs support bindings to a wide range of languages and deployment environments.

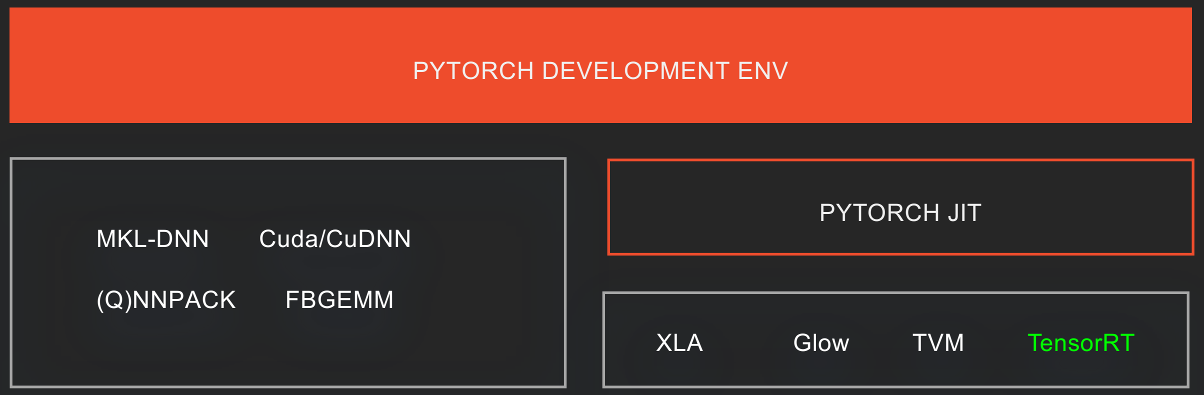
The same TorchScript models can also be loaded in the PyTorch Mobile runtime.

```
# Python: save model
traced_resnet = torch.jit.trace(torchvision.models.resnet18(),
                                torch.rand(1, 3, 224, 224))
traced_resnet.save("serialized_resnet.zip")

// C++: load model
auto module = torch::jit::load("serialized_resnet.zip");
auto example = torch::rand({1, 3, 224, 224});

// Execute `forward()` using the PyTorch JIT
auto output = module->forward({example}).toTensor();
std::cout << output.slice(1, 0, 5) << '\n';
```

OPTIMIZING FOR HARDWARE BACKENDS



TensorRT

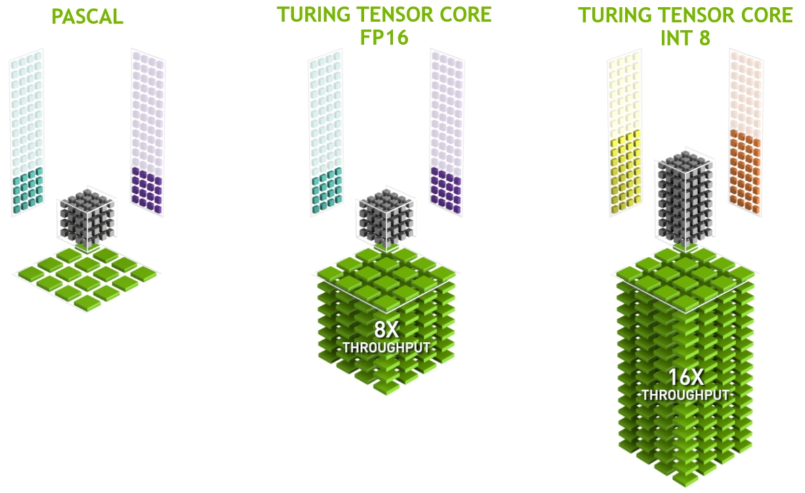
Deep learning inference optimizer and runtime that delivers low latency and high-throughput for deep learning inference applications

- Optimized for Target HW (GPU/DLA)
- Really Fast!
- Ahead of Time (AOT) compilation

Lower Precision

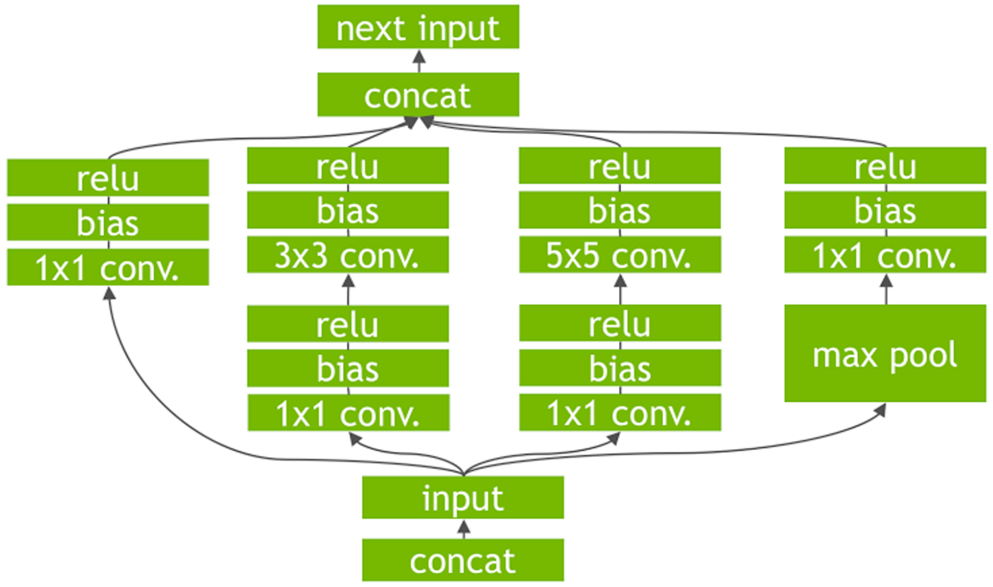
FP16 and INT8

- FP16
  - Utilizes Tensor Core Kernels on Volta and newer GPUs
- INT8
  - Post training quantization
  - Utilizes Tensor Core Kernels on Volta and newer GPUs

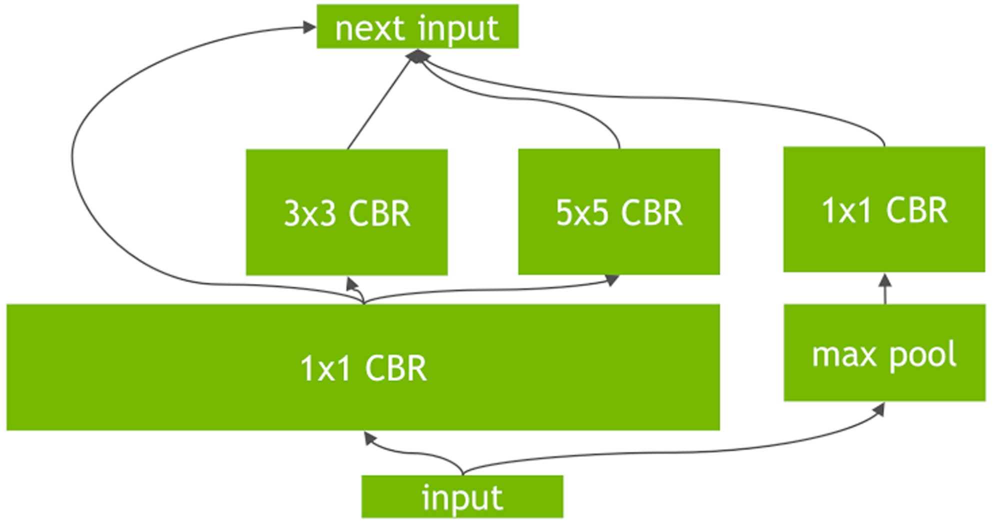


Layer & Tensor Fusion

Un-Optimized Network

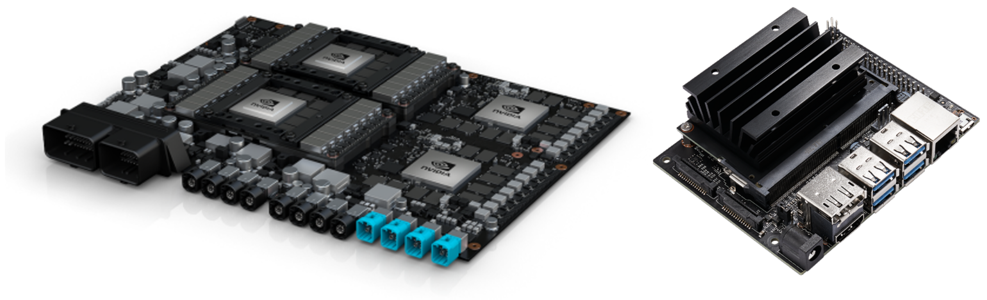


TensorRT Optimized Network



Kernel Auto Tuning

- Maximize kernel performance
- Select the best performance for target GPU
- Parameters
  - Input data size
  - Batch
  - Tensor layout
  - Input dimension
  - Memory
  - Etc.



PyTorch + TensorRT

AOT Compile PyTorch JIT?

TRTorch

TRTorch

Ahead of Time compiling of PyTorch JIT for NVIDIA GPUs

- User would either develop TorchScript code or trace PyTorch Python code to create a TorchScript Module
- Take JIT'ed PyTorch / TorchScript Module and optimize it using TensorRT

PyTorch JIT Internals

- Modules
  - The top level representation of a TorchScript program
  - Contains:
    - Parameters - e.g. weights, bias
    - Methods - e.g. forward
    - Submodules
- Methods
  - Piece of TorchScript code that takes arguments and produces an output value
  - Contains:
    - Graph - Description of the code to be run i.e. JIT IR

For more information: https://github.com/pytorch/pytorch/blob/master/torch/csrc/jit/OVERVIEW.md

Graph

TorchScript -> Graph

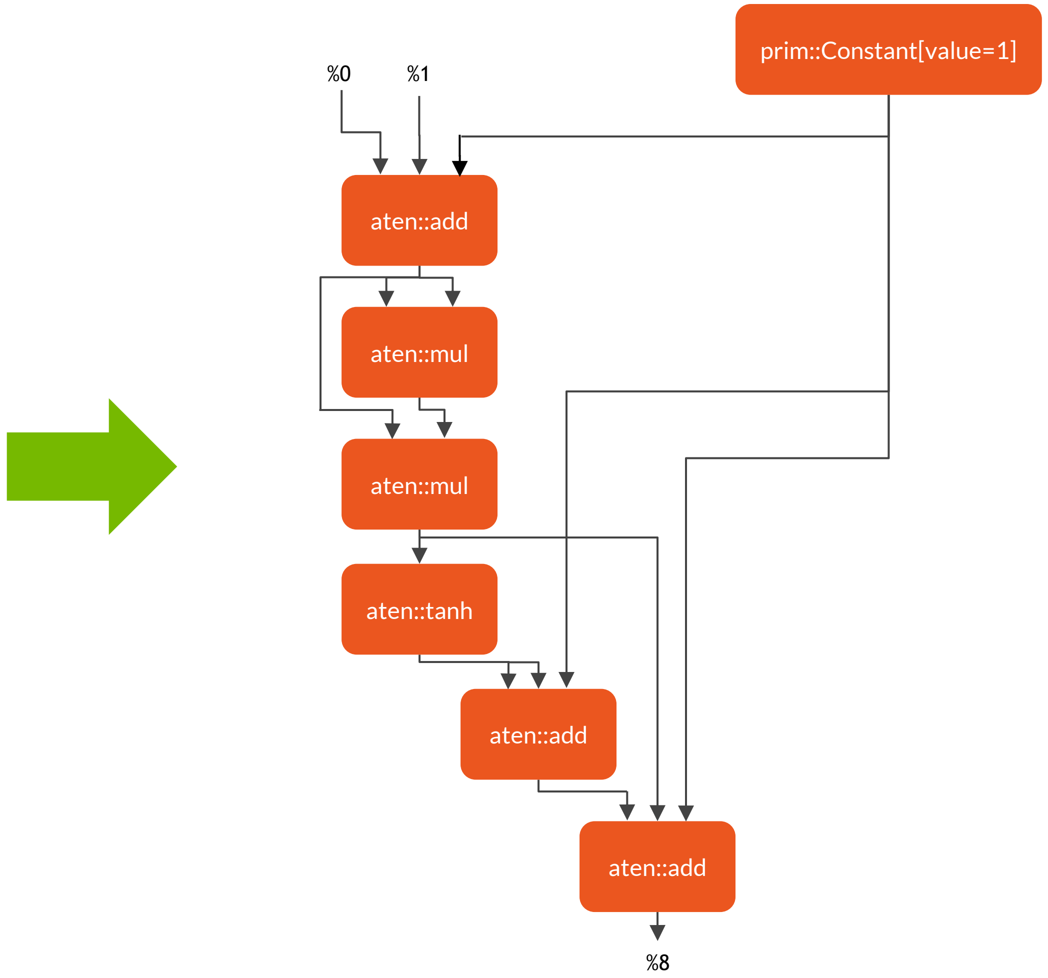
```
@torch.jit.script
def f(a, b):
  c = a + b
  d = c * c
  e = torch.tanh(d * c)
  return d + (e + e)
```

```
graph(%0 : Double(2),
      %1 : Double(2)):
  %2 : int = prim::Constant[value=1]()
  %3 : Double(2) = aten::add(%0, %1, %2)
  %4 : Double(2) = aten::mul(%3, %3)
  %5 : Double(2) = aten::mul(%4, %3)
  %6 : Double(2) = aten::tanh(%5)
  %7 : Double(2) = aten::add(%6, %6, %2)
  %8 : Double(2) = aten::add(%5, %7, %2)
  return (%8)
```

Graph

Graph -> Ops

```
graph(%0 : Tensor,
   %1 : Tensor):
 %2 : int = prim::Constant[value=1]()
 %3 : Tensor = aten::add(%0, %1, %2)
 %4 : Tensor = aten::mul(%3, %3)
 %5 : Tensor = aten::mul(%4, %3)
 %6 : Tensor = aten::tanh(%5)
 %7 : Tensor = aten::add(%6, %6, %2)
 %8 : Tensor = aten::add(%5, %7, %2)
 return (%8)
```

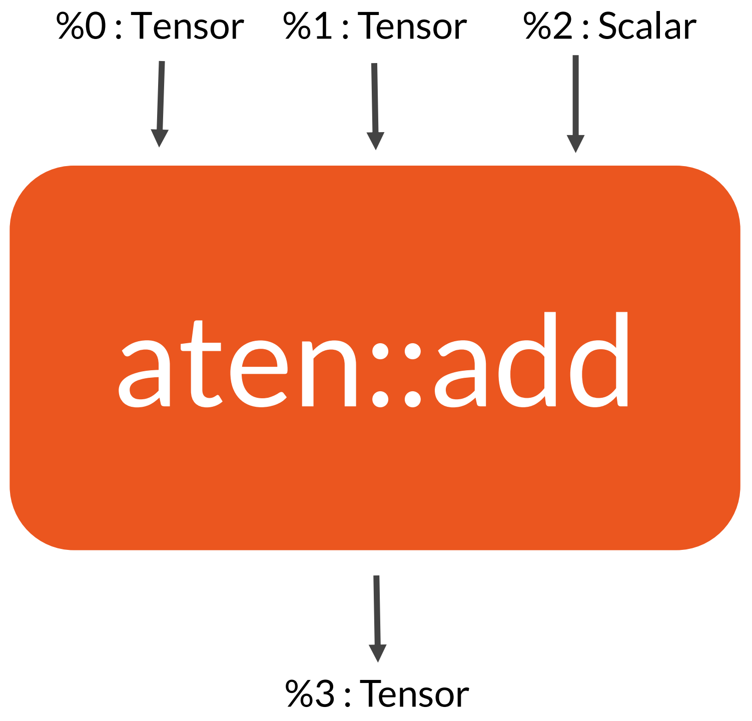


Nodes

%3 : Tensor = aten::add(%0, %1, %2)

- FunctionSchema
  - aten::add(Tensor %1, Tensor %2, Scalar %3) -> Tensor

- Values (Inputs and Outputs)
  - Values are the representation of data moving through the graph
  - Typed



Interpreter

- Stack Machine
    - State of the program execution is represented as a stack of Interpreter Values (IValues)
        - IValues contain the actual tensors, lists, etc. described abstractly by Values
    - Ops implementations consume and place IValues on the stack

TRTorch System Architecture

- Lower
  - Transform the JIT Internal Representation into one more friendly to converting to TensorRT
- Convert
  - First evaluate primitive ops (static value) and store the results
  - Add TensorRT operations to an engine builder corresponding to remaining ops
  - Return a graph with converted nodes removed and replaced with a single TRT op
- Execute
  - Instantiate the engine, when graph is run on the JIT interpreter, use the op to run the engine

Lowering

- First make a purely functional graph
- Replace subgraphs that can be friendlier to the converters (and also reduces the number of ops to write explicit converters for)

Purely Functional Graph

(PyTorch Lowering)

```
graph(%self.1 : <mangle>.Module,
      %input.1 : Tensor):
    %5 : <mangle>.Module = prim::GetAttr[name="fc1"](%self.1)
    %30 : Tensor = prim::CallMethod[name="forward"](%5, %input.1)
```

```
graph(%1 : Tensor,
      %2 : Float(10, 10),
      %3 : Float(10)):
    %res = aten::linear(%1, %2, %3)
    return (%res)
```

Graph Transforms

Transformations That Make Graphs Easier to Convert

```
graph(%1 : Tensor, %2 : int, %3 : dtype):
        %res = aten::log_softmax(%1, %2, %3)
        return (%res)
```

```
graph(%1 : Tensor, %2 : int, %3 : dtype):
         %z = aten::softmax(%1, %2, %3)
         %res = aten::log(%z)
    return (%res)
```

Conversion

- For each supported graph
    - Create a conversion context (Manages the TensorRT network definition and conversion data)
    - Add Inputs to the TensorRT NetDef based on the input tensors in the graph
    - For each node in graph
        - Collects inputs to node
            - If any inputs a result of primitive ops, evaluate the primitive op
        - With node and input Values add a corresponding node to TensorRT NetDef
    - Add Outputs

Conversion Contex

- Store of information to use during conversion, network definition, TensorRT builder infra
- Builder Resources
    - Copies of weights needed during the engine building phase
- Value Tensor Map
    - `torch::jit::Values` are an abstract representation of data flow in graph
    - `nvinfer1::ITensors` are an abstract representation of data flow in NetDef
    - map Values -> ITensors
- Evaluated Value Map
    - Stores previously evaluated nodes outputs
    - Output Value -> IValue

Evaluator

- Use the conversion context’s evaluated value map to simulate the primitive operation

- Store the output in the evaluated value map corresponding to IValue so that converters can use it to fill in layer settings

```
%4 : int = prim::Constant[value=3]()
%5 : int[] = prim::ListConstruct(%4, %4)
%6 : int[] = aten::max_pool(%in, %5, %2)
```

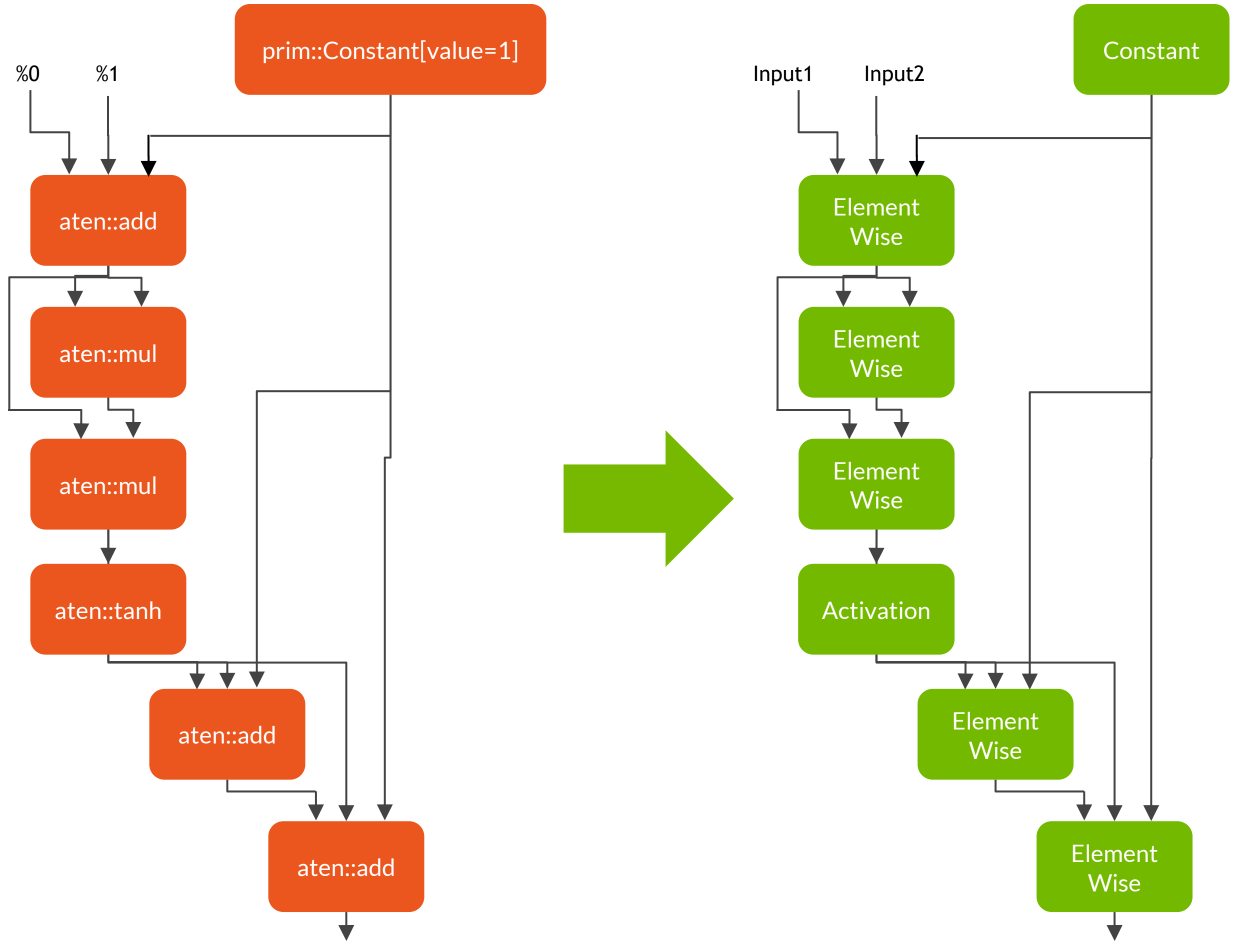
$\Rightarrow$

3

[3,3]

Converters

- Lambdas which take current context state, node and input `jit::IValues / nvinfer1::ITensors` corresponding to node values
- Adds layer corresponding to node to Network Definition
- Stores ITensor output of new layers in value tensor map
- Converters stored in global, user accessible registry



Converters

- Maps a Function Schema to Lambda
- Args are passed in Left to Right according to Function Signature
    - Union:
        - `IValue` (PyTorch Interpreter Value)
        - `ITensor` (TensorRT Abstract Tensor Representation)
- Using args add layer to `ctx->net`
- Store a mapping of PyTorch Value to TensorRT Tensor

```
auto linear_registrations = RegisterNodeConversionPatterns().pattern({
        "aten::linear(Tensor input, Tensor weight, Tensor? bias = None) -> (Tensor)",
        [](ConversionCtx* ctx, const torch::jit::Node* n, args& args) -> bool {
            auto in = args[0].ITensor();
            Weights w = Weights(ctx, args[1].unwrapToTensor());
            nvinfer1::ILayer* new_layer;
            if (args.size() == 3) {
                Weights b(ctx, args[2].unwrapToTensor());
                new_layer = ctx->net->addFullyConnected(*in, w.num_output_maps, w.data, b.data);
            } else {
                new_layer = ctx->net->addFullyConnected(*in, w.num_output_maps, w.data, {});
            }
            if (!new_layer) {
                ...
            }
            new_layer->setName(util::node_info(n).c_str());
            auto output = n->outputs()[0];
            ctx->value_tensor_map[output] = new_layer->getOutput(0);
            return true;
        }});
```

Engine Manager

- Map from Engine ID to Execution context + Function Schema for the engine

- Runs the Infer function using an execution context
    - Using PyTorch Tensors as the data management system
        - Create new Tensors to recieve engine output

Executor

- Once the converter is done, the graph that was converted is replaced with a new graph containing the JIT op which will run the corresponding to TensorRT engine

- When called in the stack, will look up the correct engine in the Execution Context Manager, pop inputs off the stack and pass them to the context manager, run the engine and push outputs onto the stack

```
void RegisterEngineOp(TRTEngine& engine) {
    torch::jit::RegisterOperators jit_registry({
            torch::jit::Operator(engine.schema,
                [engine.id](torch::jit::Stack& stack) {
                    auto io = GetEngineIO(id);
                    auto num_in = io.first;
                    auto num_out = io.second;
                    std::vector<at::Tensor> inputs;
                    for (uint64_t i = 0; i < num_in; i++) {
                        at::Tensor in;
                        torch::jit::pop(stack, in);
                        inputs.insert(inputs.begin(), std::move(in));
                    }
                    auto outputs = RunCudaEngine(GetExecCtx(id), io, inputs);
                    for (uint64_t o = 0; o < num_out; o++) {
                        torch::jit::push(stack, std::move(outputs[o]));
                    }
                    return 0;
                },aliasAnalysisFromSchema())});}
```

C++ API

- Pass `torch::jit::Module` and a module using a TensorRT engine is returned
    - Run as a normal JIT Module
- Add additional op converters to the registry to support conversion of currently unsupported or custom PyTorch Operators

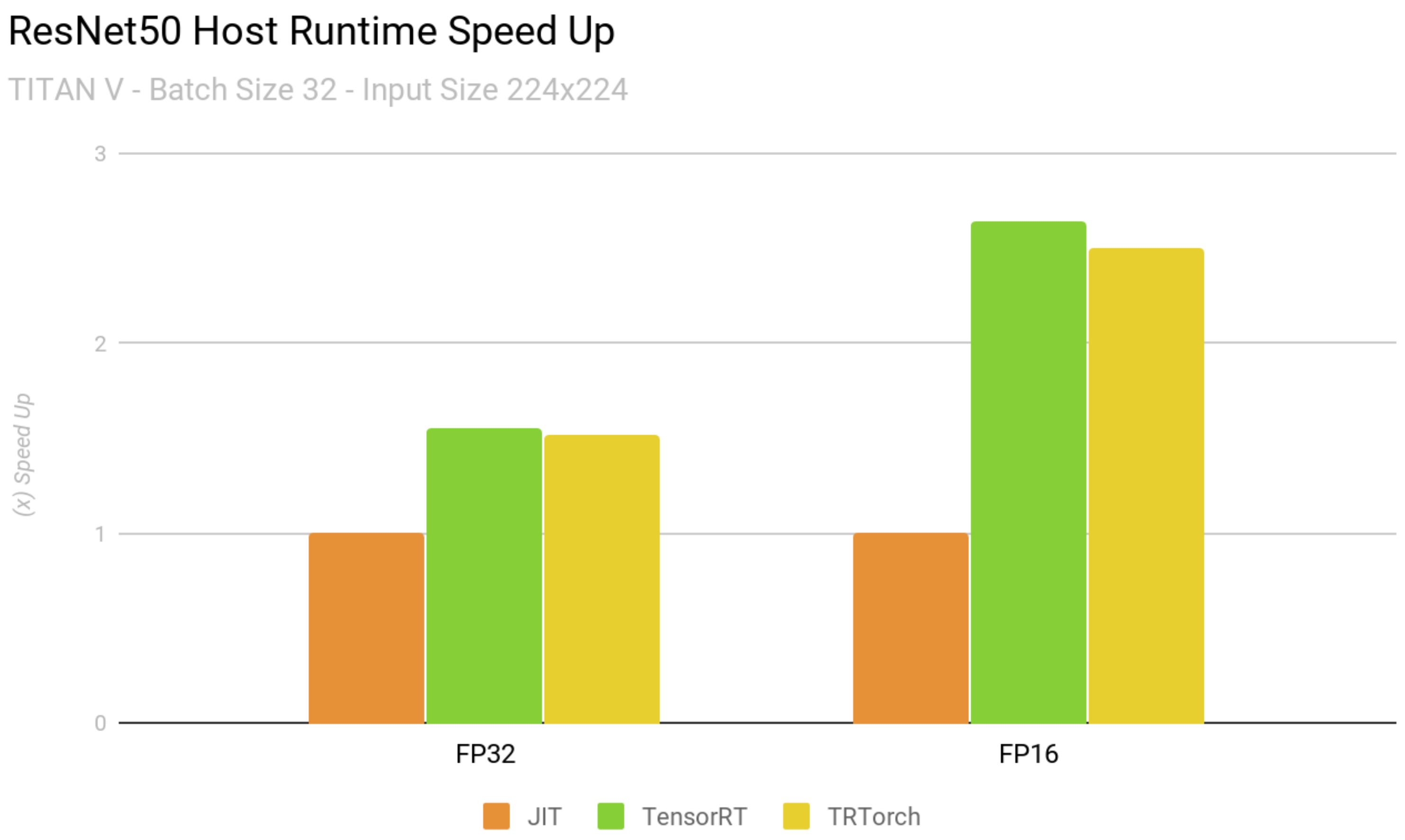
User Workflow

- Load Module
- Define expected input size (or using TensorRT dynamic input, specify input range)
- Convert
- Run

```
int main(int argc, const char* argv[]) {
    torch::jit::script::Module module;
    module = torch::jit::load(argv[1]);

    std::vector<std::vector<int64_t>> dims = {{1,1, 28, 28}};
    // or std::vector<std::vector<int64_t>> dims = {{1, 1, 16, 16},
    //                                               {1,1, 28, 28},
    //                                               {1, 1, 54, 54}};
    auto trt_mod = trtorch::CompileGraph(module, dims);
    auto results = trt_mod.forward(torch::ones({28, 28})).toTensor();
}
```

TRTorch Performance Gains



PyTorch 1.4.0 (CuDNN Benchmark mode enabled) CUDA 10.1 TensorRT 6.0.1.5, TITAN V, i7-7800X

Usecases

- Quick optimization of PyTorch Models, continue to use PyTorch infrastructure with your model and run fast inference with TensorRT

- Usable as a TensorRT Parser (like ONNX Parser) to save optimized engine to PLAN file to deploy elsewhere

- [Automotive] For objection detection / segmentation network deployment,
    - Opportunity to run backbone (like ResNet, MobileNet) using TensorRT without intermediate representation like ONNX

Roadmap

- Today
  - Early Alpha Version on Github, https://www.github.com/NVIDIA/TRTorch

    CONTRIBUTIONS ARE WELCOME!!!
    - Developing Op Converters are a great way to get started (Issues on the repository)
  - Limited set of models supported, supports basic JIT->TRT functionality, running in JIT interpreter, C++ interface
- Near Future
  - Lower precision: INT8
  - NVIDIA AGX platform for AArch64 and Deep Learning Accelerator (i.e. DLA)
- Further Out
  - TensorRT Plugins, Python API, etc.
- Working with the PyTorch Community
  - Eventually upstreaming TRTorch into PyTorch

Lower Precision

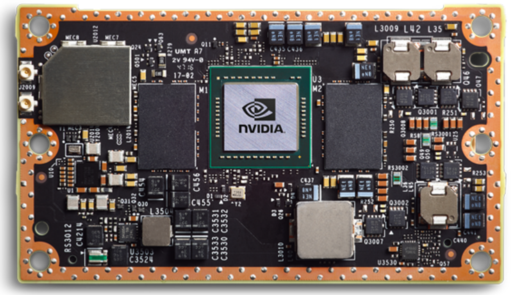
INT8

- TensorRT supports FP32, FP16, and INT8 precision
- Efficient and fast inference through INT8
    - Benefits in Memory, Power, and Computes (Tensor Core on modern GPUs)
- Leverage existing PyTorch data pipelines: Dataloader, Dataset, etc.
- Working in progress
    - Generic Torch Tensor based Int8Calibrator Implementation
    - Saving / Loading Calibration Cache
    - Internal quantization api call after calibration

NVIDIA AGX Platform

Embedded platform

- Jetson AGX
  - AArch64 & CUDA enabled SoC designed for end-to-end AI Robotics applications
  - Multiple specialized accelerators on board in Jetson Xavier
- Continue to leverage
  - PyTorch for data management and any data pipelines
  - TensorRT for graph optimizer, fast kernels, etc.
- Target compilation of PyTorch and TRTorch framework from source on AArch64



Deep Learning Accelerator (DLA)

- Fixed-function accelerator engine targeted for deep learning built into Xavier SoC
    - Convolution and Deconvolution Layers
    - Pooling Layer (Max, Average)
    - Activations (ReLU, Sigmoid, tanh)
    - Element-wise (Sum, Sub, Product, Min, Max)
    - Scale Layer
    - LRN (Local Response Normalization) Layer
    - Concat Layer

- Supports FP16 and INT8 operating precisions

- Targetable by TensorRT; anticipating GPU fallback (e.g. shuffling / transpose)

Roadmap

- Today
  - Early Alpha Version on Github, https://www.github.com/NVIDIA/TRTorch

    CONTRIBUTIONS ARE WELCOME!!!
    - Developing Op Converters are a great way to get started (Issues on the repository)
  - Limited set of models supported, supports basic JIT->TRT functionality, running in JIT interpreter, C++ interface
- Near Future
  - NVIDIA AGX platform for AArch64 and Deep Learning Accelerator (i.e. DLA)
  - Lower precision: INT8
- Further Out
  - TensorRT Plugins, Python API, etc.
- Working with the PyTorch Community
  - Eventually upstreaming TRTorch into PyTorch