TORCHSCRIPT: OPTIMIZED EXECUTION OF PYTORCH PROGRAMS

Presenter Zachary DeVito

PyTorch Design Principles

Be Pythonic A first-class member of the python ecosystem, one idiomatic way of doing things.

Put Researchers First Easy APIs for models, data loaders, and optimizers. Hide implementation complexity.

Provide Pragmatic Performance A slowdown of 10% for a simpler API is acceptable; a 2x slowdown is not

Worse is better Save time by keeping the implementation simple, and write new features instead. A simple but incomplete solution is better than a complex one that is hard to maintain

PyTorch Models are (differentiable) Python programs

```
class LinearLayer(nn.Module):
    def __init__(self, in_sz, out_sz):
        super().__init__()
        t1 = torch.randn(in_sz, out_sz)
        self.w = nn.Parameter(t1)
        t2 = torch.randn(out_sz)
        self.b = nn.Parameter(t2)

    def forward(self, activations):
        t = torch.mm(activations, self.w)
        return t + self.b
```

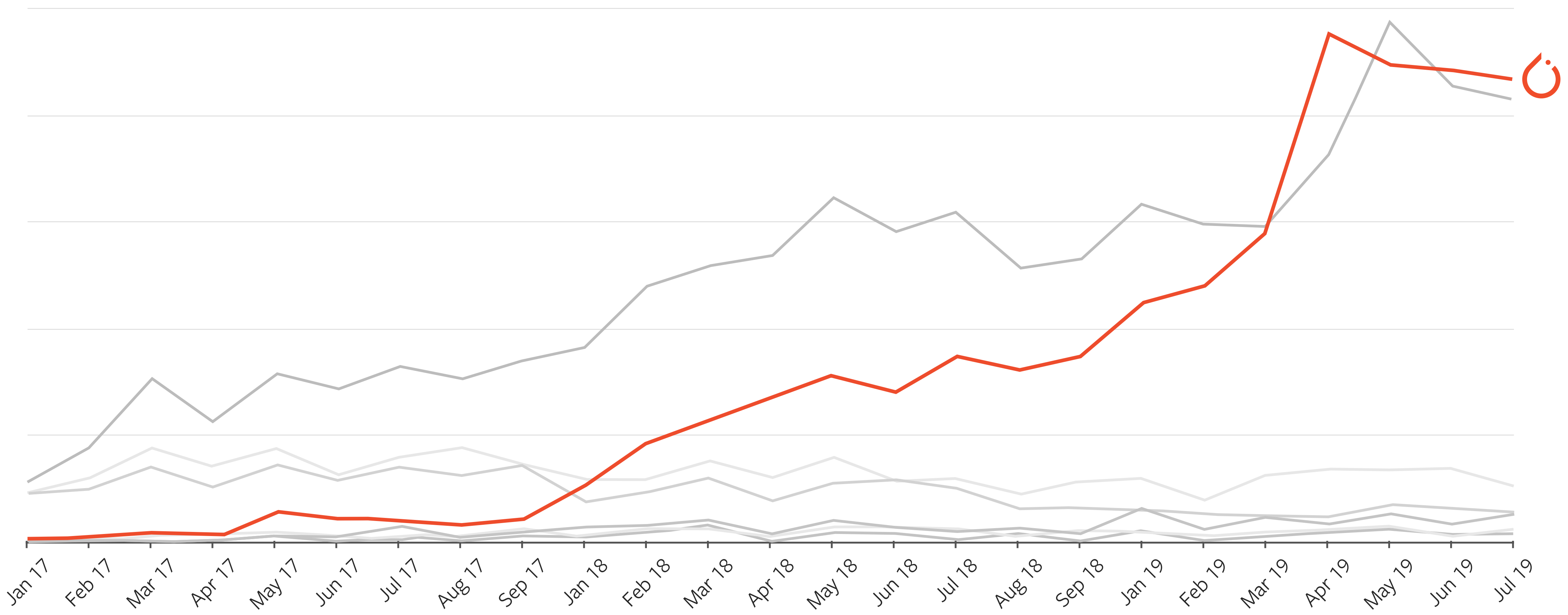
```
class FullBasicModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, 128, 3)
        self.fc = LinearLayer(128, 10)

    def forward(self, x):
        t1 = F.relu(self.conv(x))
        t2 = self.fc(t1)
        return F.softmax(t2)
```

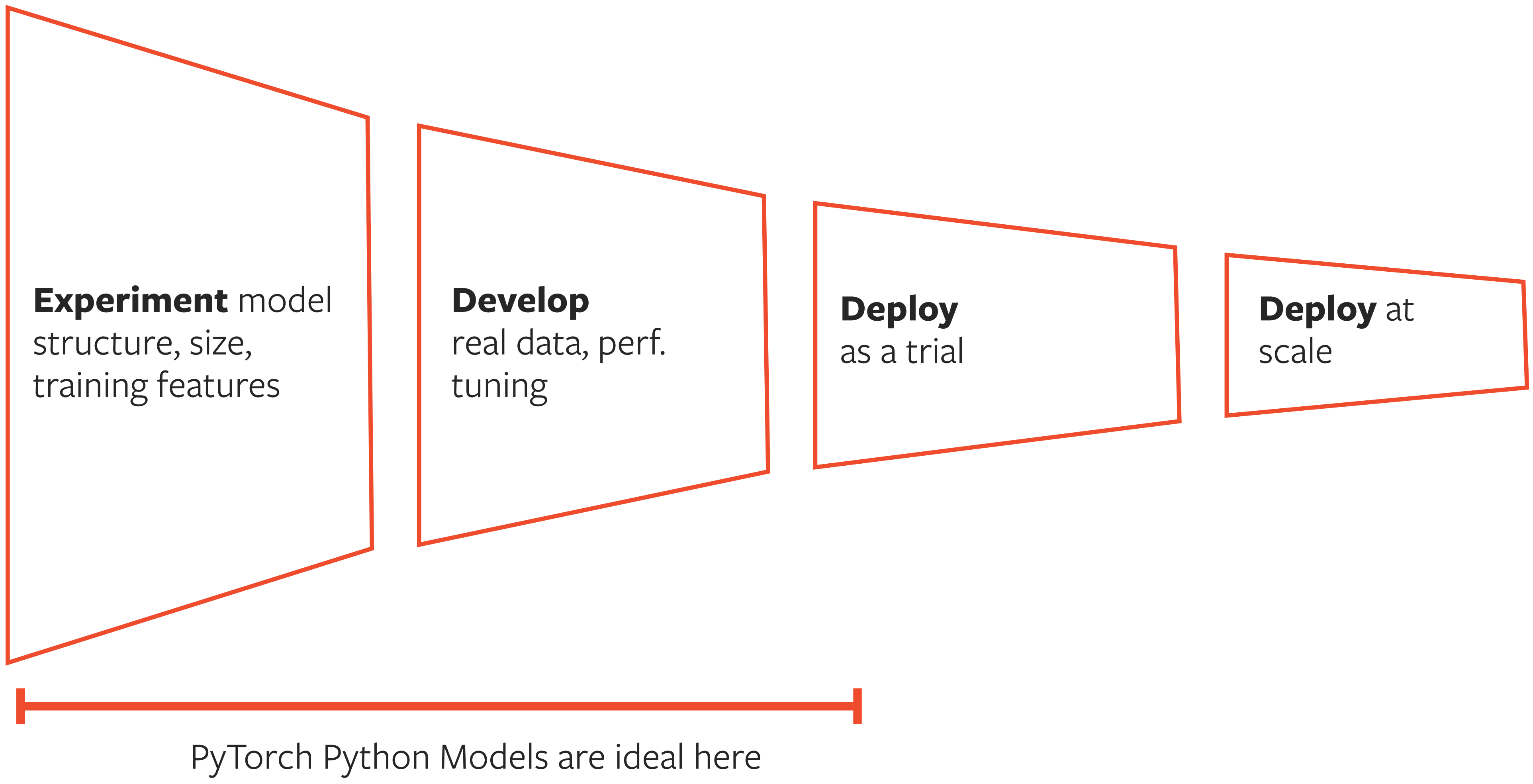
Why? Pythonic

+ Debuggable — `print` and `pdb`
+ Hackable — use any Python library

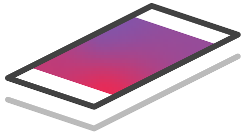
Uses well-understood object-oriented programming abstractions

GROWTH IN ARXIV MENTIONS IN RESEARCH PAPERS



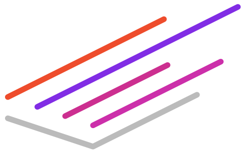

Experiment model structure, size, training features
Develop real data, perf. tuning
Deploy as a trial
Deploy at scale
PyTorch Python Models are ideal here

REQUIREMENTS FOR DEPLOYING MODELS



PORTABILITY

Models should run anywhere



PERFORMANCE

Whole-program optimization

PROBLEM STATEMENT — WE NEED A SYSTEM THAT CAN:

1

CAPTURE THE STRUCTURE OF PYTORCH PROGRAMS.

2

USE THAT STRUCTURE TO OPTIMIZE.

PROBLEM STATEMENT — WE NEED A SYSTEM THAT CAN:

1

CAPTURE THE STRUCTURE
OF PYTORCH PROGRAMS.

TORCHSCRIPT

2

USE THAT STRUCTURE
TO OPTIMIZE.

JIT COMPILER

PyTorch

Models are Python *programs*

+ Simple
+ Debuggable — `print` and `pdb`
+ Hackable — use any Python library

– Needs Python to run
– Difficult to optimize and parallelize

TorchScript

Models are ~~Python~~ *programs*,
TorchScript

an optimizable subset of Python

+ Same "models are programs" approach
+ Production deployment
+ No Python dependency
+ Optimizable

Authoring TorchScript

Write model directly in a subset of Python

- AST-driven transformation
- Control-flow is preserved
- `print` statements can be used for debugging
- Remove the annotations to debug using standard Python tools.

```
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


script_rnn = torch.jit.script(RNN(W_h, U_h, W_y, b_h, b_y))

# save the compiled code and parameters so they can run elsewhere
script_rnn.save("my_rnn.pt")
```

Loading a model without Python

Torch Script models can be saved to a model archive, and loaded in a python-free executable using a C++ API.

Our C++ Tensor API is the same as our Python API, so you can do preprocessing and post processing before calling the model.

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

// C++: load and run model
auto module = torch::jit::load("serialized_resnet.pt");
auto example = torch::rand({1, 3, 224, 224});
auto output = module.forward({example}).toTensor();
std::cout << output.slice(1, 0, 5) << '\n';
```

What subset of PyTorch is valid Torch Script?

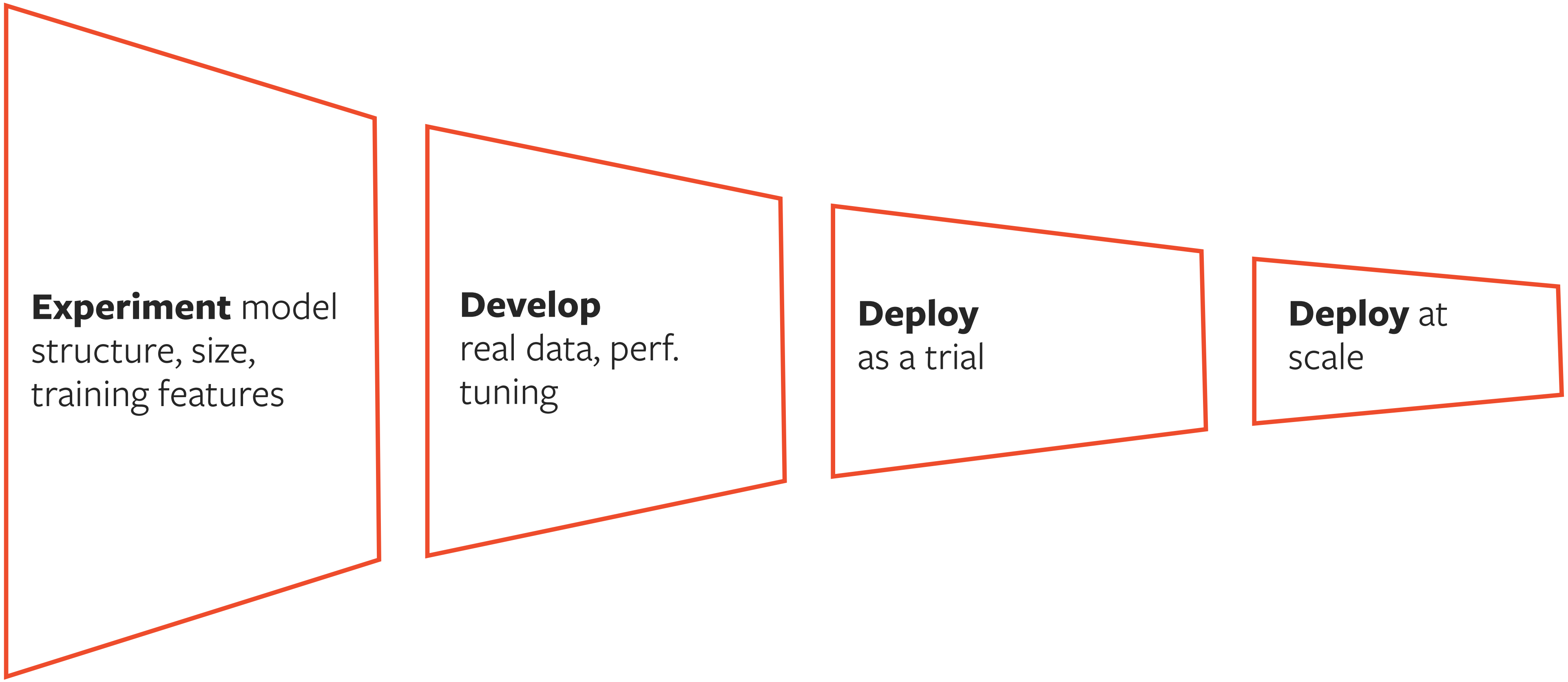
- ✓ Static typing and type inference of all values
- ✓ Tensors and numeric primitives
- ✓ If statements
- ✓ Loops (and break, continue, return)
- ✓ User-defined classes with fixed attributes
- ✓ Tuples, Lists
- ✓ `print` and strings
- ✓ Gradients propagation through script functions

- ✓ In-place updates to tensors or lists
- ✓ All standard library `nn.Module`s like `nn.Conv`

- ✗ Inheritance
- ✗ More complicated control-flow (e.g. generators)

For more details https://pytorch.org/docs/master/jit.html#torch-script-language-reference

Pay for what you use: Models only need to be in TorchScript for deployment.



Python initialization, TorchScript inference

```
# 1. Define your model
class MyMod(torch.nn.Module):
    def __init__(self):
        ...
    def forward(self):
        ...

# 2. Create an instance of your model, and run init
my_nn_module = MyMod()
# 3. Convert your model to TorchScript
my_script_module = torch.jit.script(my_nn_module)
# 4. Run inference
output = my_script_module(input)
```

Model *initialization* is Python.
Inference is TorchScript.

```python
class ResNet(torch.nn.Module):

    # Initialization code, written in Python
    def __init__(self, block, layers, num_classes=1000):
        super(ResNet, self).__init__()
        self.inplanes = 64
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)
        ...

    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                conv1x1(self.inplanes, planes * block.expansion, stride),
                nn.BatchNorm2d(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample))
        self.inplanes = planes * block.expansion
        for _ in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)
```

```python
# model code, written in TorchScript
def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

Python initialization.
TorchScript inference.

```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
```

Control flow in forward always corresponds to dynamic execution in the model

Converting nn.Modules to TorchScript

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

torch.jit.script takes a *fully initialized* nn.Module and converts it to TorchScript. The result is an instance of ScriptModule.

1. Parameters (self.weight, self.bias) are preserved
2. Submodules (self.layer1) are recursively converted
3. Attributes (self.training) are converted, *if possible*
4. Methods are converted into TorchScript, starting with the top-level module's forward method, and recursively converting any method it reaches. @torch.jit.export can set additional entry points for conversion

Model structure is *preserved* during conversion including:
Function calls, objects, control-flow, leading to accurate stack traces.

CASE STUDY

Recurrent Neural Network Grammars

December 4, 2018

Improving Semantic Parsing for Task Oriented Dialog

Conversational AI Workshop at NeurIPS 2018

By: **Arash Einolghozati**, Panupong Pasupat, **Sonal Gupta**, **Rushin Shah**, **Mrinal Mohit**, **Mike Lewis**, Luke Zettlemoyer

— Complex dynamic behavior based on the inputs
— Typically written in pure C++

```python
def forward(
    self,
    tokens: torch.Tensor,
    seq_lens: torch.Tensor,
    dict_feat: Tuple[torch.Tensor, torch.Tensor, torch.Tensor],
    actions: List[List[int]],
    contextual_token_embeddings: torch.Tensor,
    beam_size: int = 1,
    top_k: int = 1,
) -> List[Tuple[torch.Tensor, torch.Tensor]]:
    actions_idx = actions[0]
    assert len(actions_idx) > 0, "actions must be provided for training"

    token_embeddings = self.embedding(
        tokens, dict_feat, contextual_token_embeddings
    )
    beam = [self.gen_init_state(tokens, token_embeddings)]
    all_finished = False
    while not all_finished:
        # Stores plans for expansion as (score, state, action)
        plans : List[Plan] = []
        all_finished = True
        # Expand current beam states
        for state in beam:
            # Keep terminal states
            if state.finished():
                plans.append(Plan(state.neg_prob, const.TERMINAL_ELEMENT, state))
            else:
                all_finished = False
                plans.extend(self.gen_plans(state))

        beam.clear()
```

COMPLEX CONTROL FLOW

```python
    tokens, dict_feat, contextual_token_embeddings
)
beam = [self.gen_init_state(tokens, token_embeddings)]
all_finished = False
while not all_finished:
    # Stores plans for expansion as (score, state, action)
    plans : List[Plan] = []
    all_finished = True
    # Expand current beam states
    for state in beam:
        # Keep terminal states
        if state.finished():
            plans.append(Plan(state.neg_prob, const.TERMINAL_ELEMENT, state))
        else:
            all_finished = False
            plans.extend(self.gen_plans(state))

    beam.clear()
    # Take actions to regenerate the beam
    plans.sort()
    for plan in plans[:beam_size]:
        beam.append(self.execute_plan(plan, actions_idx, beam_size))

beam.sort()
res = jit.annotate(List[Tuple[torch.Tensor, torch.Tensor]], [])
for state in beam[:top_k]:
    res.append(
        (
            torch.tensor([state.predicted_actions_idx]),
            # Unsqueeze to add batch dimension
            torch.cat(state.action_scores).unsqueeze(0),
        )
    )
return res
```

USE COMMON DATA STRUCTURES

```
        tokens, dict_feat, contextual_token_embeddings
)
beam = [self.gen_init_state(tokens, token_embeddings)]
all_finished = False
while not all_finished:
    # Stores plans for expansion as (score, state, action)
    plans : List[Plan] = []
    all_finished = True
    # Expand current beam states
    for state in beam:
        # Keep terminal states
        if state.finished():
            plans.append(Plan(state.neg_prob, const.TERMINAL_ELEMENT, state))
        else:
            all_finished = False
            plans.extend(self.gen_plans(state))

    beam.clear()
    # Take actions to regenerate the beam
    plans.sort()
    for plan in plans[:beam_size]:
        beam.append(self.execute_plan(plan, actions_idx, beam_size))

beam.sort()
res = jit.annotate(List[Tuple[torch.Tensor, torch.Tensor]], [])
for state in beam[:top_k]:
    res.append(
        (
            torch.tensor([state.predicted_actions_idx]),
            # Unsqueeze to add batch dimension
            torch.cat(state.action_scores).unsqueeze(0),
        )
    )
return res
```

DEFINE YOUR OWN CLASSES

```python
@torch.jit.script
class Plan:
    def __init__(self, score: float, action: int, state: ParserState):
        self.score = score
        self.action = action
        self.state = state

     def __lt__(self, other):
        # type: (Plan) -> bool
        return self.score < other.score
```

PROBLEM STATEMENT —
WE NEED A SYSTEM THAT CAN:

1

CAPTURE THE STRUCTURE
OF PYTORCH PROGRAMS.

TORCHSCRIPT

2

USE THAT STRUCTURE
TO OPTIMIZE.

JIT COMPILER

```python
import torch
class MyModule(torch.nn.Module):
    def __init__(self, N, M, state: List[Tensor]):
        super(MyModule, self).__init__()
        self.weight = torch.nn.Parameter(torch.rand(N, M))
        self.state = state

    def forward(self, input):
        self.state.append(input)
        if input.sum() > 0:
            output = self.weight.mv(input)
        else:
            output = self.weight + input
        return output

# Compile the model code to a static representation
my_module = MyModule(3, 4, [torch.rand(3, 4)])
my_script_module = torch.jit.script(my_module)

# Save the compiled code and model data
# so it can be loaded elsewhere
my_script_module.save("my_script_module.pt")
```

TorchScript IR

```
graph(%self : ClassType<MyModule>,
      %input.1 : Tensor):
  %16 : int = prim::Constant[value=1]()
  %6 : None = prim::Constant()
  %8 : int = prim::Constant[value=0]()
  %2 : Tensor[] = prim::GetAttr[name="state"](%self)
  %4 : Tensor[] = aten::append(%2, %input.1)
  %7 : Tensor = aten::sum(%input.1, %6)
  %9 : Tensor = aten::gt(%7, %8)
  %10 : bool = aten::Bool(%9)
  %output : Tensor = prim::If(%10)
    block0():
      %11 : Tensor = prim::GetAttr[name="weight"](%self)
      %output.1 : Tensor = aten::mv(%11, %input.1)
      -> (%output.1)
    block1():
      %14 : Tensor = prim::GetAttr[name="weight"](%self)
      %output.2 : Tensor = aten::add(%14, %input.1, %16)
      -> (%output.2)
  return (%output)
```

Improving Performance with TorchScript

Standard Compiler Passes

- Dead code elimination
- Constant propagation
- Common sub-expression elimination
- Loop unrolling

Tensor Optimizations

- Algebraic peephole optimizations
- Batching of matrix multiplications
- Point-wise fusions of element-wise operations

Runtime Optimization

- No global interpreter lock (GIL)
- fork/wait parallelism at the language level

```
@torch.jit.script
def LSTMCellS(x, hx, cx, w_ih, w_hh, b_ih, b_hh):
    x_mm = x.mm(w_ih.t())
    h_mm = hx.mm(w_hh.t())
    gates = x_mm + h_mm + b_ih + b_hh
    ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)
    ingate = torch.sigmoid(ingate)
    forgetgate = torch.sigmoid(forgetgate)
    cellgate = torch.tanh(cellgate)
    outgate = torch.sigmoid(outgate)
    cy = (forgetgate * cx) + (ingate * cellgate)
    hy = outgate * torch.tanh(cy)
    return hy, cy
```

```
graph(%x : Float(*, *)
      %hx : Float(*, *)
      %cx : Float(*, *)
      %w_ih : Float(*, *)
      %w_hh : Float(*, *)
      %b_ih : Float(*)
      %b_hh : Float(*)) {
  %9 : Float(*, *) = aten::t(%w_ih)
  %10 : Float(*, *) = aten::mm(%x, %9)
  %11 : Float(*, *) = aten::t(%w_hh)
  %12 : Float(*, *) = aten::mm(%hx, %11)
  %77 : Tensor[] = prim::ListConstruct(%b_hh, %b_ih, %10, %12)
  %78 : Tensor[] = aten::broadcast_tensors(%77)
  %79 : Tensor, %80 : Tensor, %81 : Tensor, %82 : Tensor = prim::ListUnpack(%78)
  %hy : Float(*, *), %cy : Float(*, *) = prim::FusionGroup_0(%cx, %82, %81, %80, %79)
  %30 : (Float(*, *), Float(*, *)) = prim::TupleConstruct(%hy, %cy)
  return (%30);
}
```

Optimization through the dynamic behavior in Torch

```python
def linear(x: Tensor, W: Tensor, b: Tensor) -> Tensor:
    return x * W + b
```

How many dimensions does this have? *unknown*

Is this a broadcasting add? *depends on b*

Is this recording a gradient? *maybe*

Is this a square-ish matrix or a skinny one? *unknown*

Is this a float or a half? *unknown*

Challenge Broadcasting

```
def should_i_fuse(x: Tensor, y: Tensor, z: Tensor) -> Tensor:
    return x + y + z
```

Scenario 1
X : Float[1000]
Y: Float[1000]
Z: Float[1000]

✓ Fused:
2000 ops
3000 reads from memory

Unfused:
2000 ops
4000 reads from memory

Scenario 2
X : Float[3]
Y: Float[3]
Z: Float[3x1000]

Fused:
6000 ops
3006 reads from memory

✓ Unfused:
3000 ops
3006 reads from memory

Profile Guided Optimization of TorchScript

While unknown statically, properties are *very stable in practice*.

We use **profile-guided** execution of TorchScript programs with **guarded optimistic optimizations.**

```
def linear(x: Tensor, W: Tensor, b: Tensor) -> Tensor:

    return x * W + b
```

How many dimensions does this have? *1.*

Is this a broadcasting add? *yes.*

Is this recording a gradient? *no.*

Is this a float or a half? *float.*

Is this a square-ish matrix or a skinny one? *square-ish.*

Example: Profile Guided Optimization of TorchScript

```
def forward(self, input : Tensor, state : Tuple[Tensor, Tensor])
-> Tuple[Tensor, Tuple[Tensor, Tensor]]:
    hx, cx = state
    gates = (torch.mm(input, self.weight_ih.t()) + self.bias_ih +
             torch.mm(hx, self.weight_hh.t()) + self.bias_hh)
    ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

    ingate = torch.sigmoid(ingate)
    forgetgate = torch.sigmoid(forgetgate)
    cellgate = torch.tanh(cellgate)
    outgate = torch.sigmoid(outgate)

    cy = (forgetgate * cx) + (ingate * cellgate)
    hy = outgate * torch.tanh(cy)

    return hy, (hy, cy)
```

```
graph(%self : __torch__.LSTMCell,
    %input.1 : Tensor,
    %state.1 : (Tensor, Tensor)):
%23 : int = prim::Constant[value=4]() # ../eg.py:24:60
%24 : int = prim::Constant[value=1]() # ../eg.py:24:63
%hx.1 : Tensor, %cx.1 : Tensor = prim::TupleUnpack(%state.1)
%7 : Tensor = prim::GetAttr[name="weight_ih"](%self)
%8 : Tensor = aten::t(%7) # ../eg.py:22:33
%9 : Tensor = aten::mm(%input.1, %8) # ../eg.py:22:17
%10 : Tensor = prim::GetAttr[name="bias_ih"](%self)
%12 : Tensor = aten::add(%9, %10, %24) # ../eg.py:22:17
%14 : Tensor = prim::GetAttr[name="weight_hh"](%self)
%15 : Tensor = aten::t(%14) # ../eg.py:23:30
%16 : Tensor = aten::mm(%hx.1, %15) # ../eg.py:23:17
%18 : Tensor = aten::add(%12, %16, %24) # ../eg.py:22:17
%19 : Tensor = prim::GetAttr[name="bias_hh"](%self)
%gates.1 : Tensor = aten::add(%18, %19, %24) # ../eg.py:22:1
...
```

Code in bold is fusible but we must know it runs on the GPU, it is floating point, and how tensors its are broadcast

Example: Profile Guided Optimization of TorchScript

```
graph(%self : __torch__.LSTMCell,
    %input.1 : Tensor,
    %state.1 : (Tensor, Tensor)):
%4 : int = prim::Constant[value=1]() # ../eg.py:24:63
%hx.1 : Tensor, %cx.1 : Tensor = prim::TupleUnpack(%state.1)
%7 : Tensor = prim::GetAttr[name="weight_ih"](%self)
%8 : Tensor = prim::profile(%7)
%9 : Tensor = aten::t(%8) # ../eg.py:22:33
%10 : Tensor = prim::profile(%input.1)
%11 : Tensor = prim::profile(%9)
%12 : Tensor = aten::mm(%10, %11) # ../eg.py:22:17
%13 : Tensor = prim::GetAttr[name="bias_ih"](%self)
%14 : Tensor = prim::profile(%12)
%15 : Tensor = prim::profile(%13)
%16 : Tensor = aten::add(%14, %15, %4) # ../eg.py:22:17
%17 : Tensor = prim::GetAttr[name="weight_hh"](%self)
%18 : Tensor = prim::profile(%17)
%19 : Tensor = aten::t(%18) # ../eg.py:23:30
%20 : Tensor = prim::profile(%hx.1)
%21 : Tensor = prim::profile(%19)
%22 : Tensor = aten::mm(%20, %21) # ../eg.py:23:17
%23 : Tensor = prim::profile(%16)
%24 : Tensor = prim::profile(%22)
%25 : Tensor = aten::add(%23, %24, %4) # ../eg.py:22:17
%26 : Tensor = prim::GetAttr[name="bias_hh"](%self)
%27 : Tensor = prim::profile(%25)
%28 : Tensor = prim::profile(%26)
%gates.1 : Tensor = aten::add(%27, %28, %4) # ../eg.py:22:17
...
```

(1) Insert profiling code at every use of a Tensor

Example: Profile Guided Optimization of TorchScript

```
graph(%self : __torch__.LSTMCell,
      %input.1 : Tensor,
      %state.1 : (Tensor, Tensor)):
  %3 : int = prim::Constant[value=4]() # ../eg.py:24:60
  %4 : int = prim::Constant[value=1]() # ../eg.py:24:63
  %hx.1 : Tensor, %cx.1 : Tensor = prim::TupleUnpack(%state.1)
  %7 : Tensor = prim::GetAttr[name="weight_ih"](%self)
  %8 : Float(40, 10) = prim::profile(%7)
  %9 : Tensor = aten::t(%8) # ../eg.py:22:33
  %10 : Float(8, 10) = prim::profile(%input.1)
  %11 : Float(10, 40) = prim::profile(%9)
  %12 : Tensor = aten::mm(%10, %11) # ../eg.py:22:17
  %13 : Tensor = prim::GetAttr[name="bias_ih"](%self)
  %14 : Float(8, 40) = prim::profile(%12)
  %15 : Float(40) = prim::profile(%13)
  %16 : Tensor = aten::add(%14, %15, %4) # ../eg.py:22:17
  %17 : Tensor = prim::GetAttr[name="weight_hh"](%self)
  %18 : Float(40, 10) = prim::profile(%17)
  %19 : Tensor = aten::t(%18) # ../eg.py:23:30
  %20 : Float(8, 10) = prim::profile(%hx.1)
  %21 : Float(10, 40) = prim::profile(%19)
  %22 : Tensor = aten::mm(%20, %21) # ../eg.py:23:17
  %23 : Float(8, 40) = prim::profile(%16)
  %24 : Float(8, 40) = prim::profile(%22)
  %25 : Tensor = aten::add(%23, %24, %4) # ../eg.py:22:17
  %26 : Tensor = prim::GetAttr[name="bias_hh"](%self)
  %27 : Float(8, 40) = prim::profile(%25)
  %28 : Float(40) = prim::profile(%26)
  %gates.1 : Tensor = aten::add(%27, %28, %4) # ../eg.py:22:17
  ...
```

(2) Run the graph a few times to record sizes

Example: Profile Guided Optimization of TorchScript

```
graph(%self : __torch__.LSTMCell,
    %input.1 : Tensor,
    %state.1 : (Tensor, Tensor)):
    %3 : int = prim::Constant[value=4]() # ../eg.py:24:60
    %4 : int = prim::Constant[value=1]() # ../eg.py:24:63
    %hx.1 : Tensor, %cx.1 : Tensor = prim::TupleUnpack(%state.1)
    %7 : Tensor = prim::GetAttr[name="weight_ih"](%self)
    %8 : Float(40, 10) = prim::guard(%7)
    %9 : Tensor = aten::t(%8) # ../eg.py:22:33
    %10 : Float(8, 10) = prim::guard(%input.1)
    %11 : Float(10, 40) = prim::guard(%9)
    %12 : Tensor = aten::mm(%10, %11) # ../eg.py:22:17
    ...
```

guard failure!

```
Unoptimized Fallback
%15 : Tensor = aten::t(%14) # ../eg.py
%16 : Tensor = aten::mm(%hx.1, %15) #
%18 : Tensor = aten::add(%12, %16, %24
...
```

(3) Replace profile nodes with guards. If a guard fails during execution, we fallback to the unoptimized code.

```
graph(%self : __torch__.LSTMCell,
      %input.1 : Tensor,
      %state.1 : (Tensor, Tensor)):
  ...
  %98 : Float(40) = prim::guard(%26, %25, %93)
  %122 : Tensor[] = prim::ListConstruct(%25, %98)
  %123 : Tensor[] = aten::broadcast_tensors(%122)
  %124 : Tensor, %125 : Tensor = prim::ListUnpack(%123)
  %hy.1 : Float(8, 10), %cy.1 : Float(8, 10) = prim::FusionGroup_1(%93, %125, %124)
  %60 : (Tensor, Tensor) = prim::TupleConstruct(%hy.1, %cy.1)
  %62 : (Tensor, (Tensor, Tensor)) = prim::TupleConstruct(%hy.1, %60)
  return (%62)

with prim::FusionGroup_1 = graph(%13 : Float(8, 10),
      %39 : Tensor,
      %44 : Tensor):
  %45 : Float(8, 10), %46 : Float(8, 10), %47 : Float(8, 10), %48 : Float(8, 10) = prim::ConstantChunk[chunks=4, dim=1
  %40 : Float(8, 10), %41 : Float(8, 10), %42 : Float(8, 10), %43 : Float(8, 10) = prim::ConstantChunk[chunks=4, dim=1
  %37 : int = prim::Constant[value=1]() # ../eg.py:24:63
  %38 : Float(8, 10) = aten::add(%45, %40, %37)
  %34 : Float(8, 10) = aten::add(%46, %41, %37)
  %30 : Float(8, 10) = aten::add(%47, %42, %37)
  %26 : Float(8, 10) = aten::add(%48, %43, %37)
  %ingate.3 : Float(8, 10) = aten::sigmoid(%38) # ../eg.py:26:17
  %forgetgate.3 : Float(8, 10) = aten::sigmoid(%34) # ../eg.py:27:21
  %cellgate.3 : Float(8, 10) = aten::tanh(%30) # ../eg.py:28:19
  %outgate.3 : Float(8, 10) = aten::sigmoid(%26) # ../eg.py:29:18
  ...
  return (%hy.1, %cy.1)
```

(4) Remove redundant guards, and use profiled properties to apply fusion.

Profile-guided optimization

Possibilities

If we know tensors are constant, we can pre-multiplying weights to remove batch norms or load weights into grams on accelerators.

If we know that the bool that is input to an if-statement is almost always true, we can eliminate the other branch from the optimized code.

WHAT'S NEXT?

TORCHSCRIPT AS A PLATFORM

QUANTIZATION

Model quantization done safely and automatically using JIT transformations.

MOBILE

A lightweight interpreter that can run on-device.

BACKENDS

Support for lowering models to static graph compilers, like TVM, Glow, XLA.

TRY IT

AND GIVE US FEEDBACK!

TUTORIALS

pytorch.org/tutorials

Introduction to TorchScript:
https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html

Loading a TorchScript model in C++:
https://pytorch.org/tutorials/advanced/cpp_export.html

DOCUMENTATION

TorchScript reference:
https://pytorch.org/docs/master/jit.html

FEEDBACK

"jit" label on github:
https://github.com/pytorch/pytorch/issues?q=is%3Aissue+is%3Aopen+label%3Ajit