HOW USABILITY IMPROVES PERFORMANCE IN PYTORCH

PRESENTED BY ZACHARY DEVITO
Why was PyTorch successful?

- Performance? No, original design allowed 20% slowdown for a better API.
Why was PyTorch successful?

Laser-focused on usability
Eager mode by default

A graph is created on the fly

\[
W_{h} = \text{torch.randn}(20, 20, \text{requires_grad=True}) \\
W_{x} = \text{torch.randn}(20, 10, \text{requires_grad=True}) \\
x = \text{torch.randn}(1, 10) \\
prev_{h} = \text{torch.randn}(1, 20)
\]

+ Bindings for SOTA algorithms: CUDNN, BLAS, Intel MKL
LASER FOCUSED ON USABILITY

< 24 hour response time on GitHub issues and forums
At the time, competitors gained little with graph-mode but did reduce usability.

<table>
<thead>
<tr>
<th>Framework</th>
<th>AlexNet</th>
<th>VGG-19</th>
<th>ResNet-50</th>
<th>MobileNet</th>
<th>GNMTv2</th>
<th>NCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chainer</td>
<td>778 ± 15</td>
<td>N/A</td>
<td>219 ± 1</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CNTK</td>
<td>845 ± 8</td>
<td>84 ± 3</td>
<td>210 ± 1</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MXNet</td>
<td>1554 ± 22</td>
<td>113 ± 1</td>
<td>218 ± 2</td>
<td>444 ± 2</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>PaddlePaddle</td>
<td>933 ± 123</td>
<td>112 ± 2</td>
<td>192 ± 4</td>
<td>557 ± 24</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>1422 ± 27</td>
<td>66 ± 2</td>
<td>200 ± 1</td>
<td>216 ± 15</td>
<td>9631 ± 1.3%</td>
<td>4.8e6 ± 2.9%</td>
</tr>
<tr>
<td>PyTorch</td>
<td>1547 ± 316</td>
<td>119 ± 1</td>
<td>212 ± 2</td>
<td>463 ± 17</td>
<td>15512 ± 4.8%</td>
<td>5.4e6 ± 3.4%</td>
</tr>
</tbody>
</table>

**Table 1**: Training speed for 6 models using 32bit floats. Throughput is measured in images per second for the AlexNet, VGG-19, ResNet-50, and MobileNet models, in tokens per second for the GNMTv2 model, and in samples per second for the NCF model. The fastest speed for each model is shown in bold.

Productivity vs (?) Performance
Productivity enables Performance

44x less compute required to get to AlexNet performance 7 years later (linear scale)

The Usability Crisis of Accelerators
Don’t compromise usability for potential performance gains.

Empower users to fix any potential performance issues with incrementally increasing effort.
THIS TALK

01
CASE STUDY: FIXED SIZES AND USABILITY

02
UPCOMING TOOLS IN PYTORCH FOR USABILITY
REAL NETWORKS DO NOT ALWAYS HAVE FIXED SIZES

... BUT MANY LIBRARIES DO
Vision: Images are not the same size, but batches are rectilinear

Scaling - have to check accuracy

Padding - wastes compute
But conv is the same at each pixel

Should the conv primitive have a non-rectilinear batch instead?
Text: Sequences are not the same length

[“The”, “sun”, “is”, “very”, “bright”]
[“Let’s”, “be”, “friends”]
[“It”, “was”, “for”, “you”]

Transforms have lots of per-word arithmetic
Effective Transformer:
flattens/unflattens for per-word operations

```c
__global__ void compress_bert_input(
    const __half* from_tensor, const int* mask, const int* prefix_sum,
    __half* to_tensor, int* batch_idx, int* word_idx,
    int batch_size, int seq_len, int hidden_dim) {
    int bid = blockIdx.y;  // batch
    int wid = blockIdx.x;  // word
    int tid = threadIdx.x;  //

    // 1. count pos for from tensor
    int mask_idx = bid * seq_len + wid;
    if (mask[mask_idx] > 0.5) {
        int valid_idx = prefix_sum[mask_idx];

        // 2. write batch id and word id for each word
        if (tid == 0) {
            batch_idx[valid_idx] = bid;
            word_idx[valid_idx] = wid;
        }

        // 3. copy src data
        half2* src_ptr = (half2*)from_tensor;
        half2* dst_ptr = (half2*)to_tensor;
        int src_idx = mask_idx * hidden_dim + tid;
        int dst_idx = valid_idx * hidden_dim + tid;
        dst_ptr[dst_idx] = src_ptr[src_idx];
    }
}
```

https://github.com/bytedance/effective_transformer
Effective Transformer:
flatten/unflatten for per-word operations

Tesla V100, float16, maximum sequence length=32, average sequence length≈20

<table>
<thead>
<tr>
<th>batch_size</th>
<th>XLA (in ms)</th>
<th>Faster Transformer (in ms)</th>
<th>Speedup over XLA</th>
<th>Effective Transformer (in ms)</th>
<th>Speedup over XLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>15.08</td>
<td>10.39</td>
<td>1.45</td>
<td>8.75</td>
<td>1.72</td>
</tr>
<tr>
<td>200</td>
<td>28.08</td>
<td>19.64</td>
<td>1.43</td>
<td>15.32</td>
<td>1.83</td>
</tr>
<tr>
<td>300</td>
<td>41.37</td>
<td>29.65</td>
<td>1.40</td>
<td>22.18</td>
<td>1.86</td>
</tr>
<tr>
<td>400</td>
<td>53.65</td>
<td>38.52</td>
<td>1.39</td>
<td>28.31</td>
<td>1.89</td>
</tr>
<tr>
<td>500</td>
<td>66.86</td>
<td>48.13</td>
<td>1.39</td>
<td>33.08</td>
<td>2.02</td>
</tr>
<tr>
<td>1000</td>
<td>131.46</td>
<td>95.01</td>
<td>1.38</td>
<td>64.34</td>
<td>2.04</td>
</tr>
</tbody>
</table>

https://github.com/bytedance/effective_transformer
It is harder to add custom behavior in deeper stack

4 custom APIs to understand:
- Optimistic case: possible, but can degrade optimization techniques
- Pessimistic case: layer doesn’t allow for a the custom op, user is blocked.
Vocab Reduction: varying vocab size

\[ v_{\text{inference}} = \{ v_i \text{ if } \text{score}(v_i, \text{input_tokens}) > \text{threshold} \} \]

CPU decoding up to 10x faster.

A surprising amount of dynamic behavior and sizes occur in real world models.

So when is it ok to restrict dynamic behavior to get better performance?
— Add restrictions when there are already-realized performance gains. (e.g. 30% slower without using half precision)
— When there are theoretical gains (e.g. if sizes are known we can allocate memory ahead of time, and do layout planning), err on the side of usability.

Why? There are 10x more external than internal developers, and each restriction prohibits them from exploring application-specific optimizations or adds friction
USABILITY IN PYTORCH FOR PRODUCTION
USER STUDY OF TORCHSCRIPT USERS

— Capture the structure of PyTorch programs to do custom transforms
— Create self-contained archives of trained PyTorch programs for transfer learning, or to deploy in a production environment
— Serve models as part of a service (e.g. a language translation server)
— Improve the performance of these models

Can we decouple these uses to make each task easier?
class MyModule(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.param = torch.nn.Parameter(torch.rand(3, 4))
        self.linear = torch.nn.Linear(4, 5)

    def forward(self, x):
        return self.linear(x + self.param).clamp(min=0.0, max=1.0)

module = MyModule()

from torch.fx import symbolic_trace, GraphModule
# Symbolic tracing frontend - captures the semantics of the module
symbolic_traced: GraphModule = symbolic_trace(module)

# High-level intermediate representation (IR) - Graph representation
print(symbolic_traced.graph)

   graph(x):
     %param : [#users=1] = self.param
     %add_1 : [#users=1] = call_function[target=<built-in function add>](args = (%x, %param), kwargs = {})
     %linear_1 : [#users=1] = call_module[target=linear](args = (%add_1,), kwargs = {})
     %clamp_1 : [#users=1] = call_method[target=clamp](args = (%linear_1,), kwargs = {min: 0.0, max: 1.0})
     return clamp_1
**OVERVIEW**

**torch.fx**

**SYMBOLIC TRACING**

Construct IR by tracing the execution of a PyTorch model, similar to TF Autograph and JAX’s jaxpr tracing.

**SIMPLE HIGH-LEVEL IR**

4 instruction IR that represents PyTorch programs using the publicly documented PyTorch operators.

**PYTHON CODE GENERATION**

After you are done working with the IR, you can transform it back to Python code and use it in eager mode, or pass it TorchScript to improve performance.
torch.package self-contained eager-mode models

```python
from torch.package import PackageExporter

model, example_input = load_tacotron2()

with PackageExporter('tacotron2') as e:
    # Configure how to export the source code
    e.extern(['torch.*'])
    # instead of saving the source code for the
    # torch libraries, let the package link to
    # the libraries of the loading process.

    # Replace these modules with mock
    # implementations. They are not
    # actually used.
    e.mock(['numpy', 'librosa.*', 'scipy.*'])

from torch.package import PackageImporter

i = PackageImporter('tacotron2')

i.save_pickle('model', 'model.pkl', model)
# code for the model is loaded from the model_file
# rather than the normal import system, except
# where packages are marked as extern.
model = i.load_pickle('model', 'model.pkl')
example_inputs = i.load_pickle('model', 'eg.pkl')
# test the model
model(*example_inputs)
```

(a) Packaged model structure  (b) Model export  (c) Model import

[https://arxiv.org/abs/2104.00254]
torch::deploy  a native library for running packaged models

[https://arxiv.org/abs/2104.00254]
PICK THE RIGHT TOOLS FOR YOUR PROBLEM

— Capture the structure of PyTorch programs: torch.fx
— Create self-contained archives of trained PyTorch programs: torch.package
— Serve models as part of a service: torch::deploy
— Improve the performance of these models: TorchScript

... AND ADD YOUR OWN TOOLS TO THE ECOSYSTEM

— e.g use torch.package to save a PyTorch model that uses TVM to construct custom operators
— e.g use torch.fx to extract a PyTorch program, and write a transformer to run it on new accelerator hardware.
Keep framework users in the loop about hardware performance, empower them to fix performance issues with incrementally increasing effort.

As part of PyTorch, we are trying to build tools to increase usability and lower the friction of getting models into production. Let us know how we can help.
RESOURCES


SPECIAL THANKS TO PYTORCH CONTRIBUTORS WORKING ON OUR USABILITY PROJECTS: JASON ANSEL, WILL CONSTABLE, HORACE HE, JAMES REED, MICHAEL SUO, ANSLEY USSERTY, AILING ZHANG