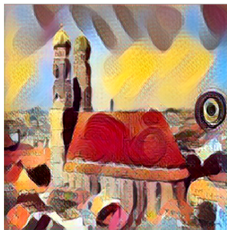
PYTORCH

PyTorch in Munich Meetup

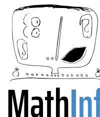


PyTorch, JIT, Android

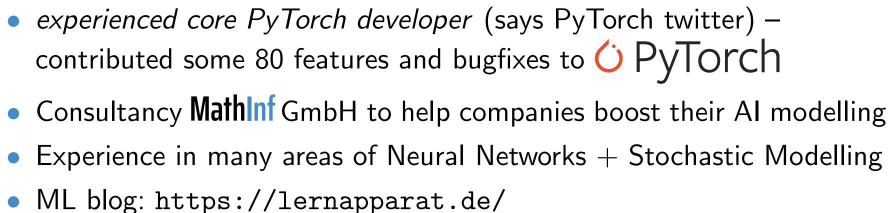
Thomas Viehmann
tv@mathinf.eu

PyTorch in Munich @ Microsoft, 11 December 2018

About me

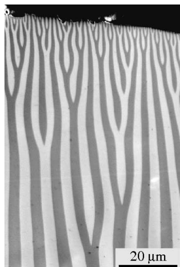


Thomas Viehmann (@tom on PyTorch, @t-vi on Github)



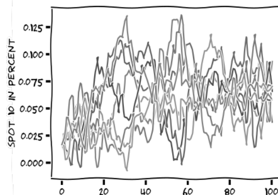
- *experienced core PyTorch developer* (says PyTorch twitter) – contributed some 80 features and bugfixes to PyTorch
- Consultancy MathInf GmbH to help companies boost their AI modelling
- Experience in many areas of Neural Networks + Stochastic Modelling
- ML blog: `https://lernapparat.de/`

My background besides AI

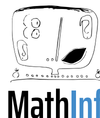


- Mathematical modeller
- Ph.D. in Mathematics (Bonn) – Mathematical proof of fractal behaviour in a model for magnets

- Actuary and consultant for 9 years - helping insurance companies with their maths for financial and risk modelling, statistics etc.



Thanks!



PYTORCH

I like PyTorch the software... but to me the best part are the people

- on the forums
- here!

I'm indebted to many PyTorch people for advice and encouragement for those bits that I worked on:
Adam Paszke, Francisco Massa, Natalia Gimelshein, Peter Goldsborough, Piotr Bialecki, Simon Wang, Soumith Chintala and many others. Thanks!

(But errors and opinions are my own and not theirs.)

PyTorch – Pythonic deep learning

MathInf

...from the *Fast neural style transfer*[1] example (do check it out)!

```
class ResidualBlock(torch.nn.Module):
  def __init__(self, channels):
      super().__init__()
      self.conv1 = ConvLayer(channels, channels, kernel_size=3, stride=1)
      self.in1 = torch.nn.InstanceNorm2d(channels, affine=True)
      self.conv2 = ConvLayer(channels, channels, kernel_size=3, stride=1)
      self.in2 = torch.nn.InstanceNorm2d(channels, affine=True)
      self.relu = torch.nn.ReLU()

  def forward(self, x):
      residual = x
      out = self.relu(self.in1(self.conv1(x)))
      out = self.in2(self.conv2(out))
      out = out + residual
      return out
```

[1] https://github.com/pytorch/examples/tree/master/fast_neural_style

PyTorch – Pythonic deep learning

MathInf

The Zen Of Python: Explicit is better than Implicit

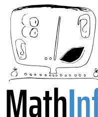
```
transformer = TransformerNet().to(device)
optimizer = Adam(transformer.parameters(), args.lr)
mse_loss = torch.nn.MSELoss()

for e in range(args.epochs):
    for batch_id, (x, _) in enumerate(train_loader):
        optimizer.zero_grad()
        x = x.to(device)
        y = transformer(x)
        features_y = vgg(y)
        features_x = vgg(x)
        content_loss = args.content_weight * mse_loss(features_y.relu2_
        style_loss = ...

        total_loss = content_loss + style_loss
        total_loss.backward()
        optimizer.step()
```

...but Ignite and fast.ai have a great `.fit`, too.

The PyTorch JIT

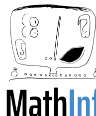


But what if you want to go beyond Python?

- for speed
- for deployment

The PyTorch JIT (=Just In Time compiler) to the rescue!

The PyTorch JIT



Two ways to specify TorchScript (=JIT) programs:

tracing

Watch me! – Now do the same!
(recording)

- Can use any Python...
- ...but the JIT wont (try to) understand it all.
- Only Tensors and Tensor functions are recorded.

```
def myfn(x):
  for i in range(5):
    x = x * x
  return x
a = torch.randn(5)
traced_fn = torch.jit.trace(myfn,(a,))
traced_fn(a)
```

scripting

Here is how. – Now do it!
(classical programming)

- The JIT will (try to) understand all code...
- ...but can't use all of Python.
- Focus on typical subset, including for loops, if, ...

```
@torch.jit.script
def script_fn(x):
  for i in range(5):
    x = x * x
  return x
a = torch.randn(5)
script_fn(a)
```

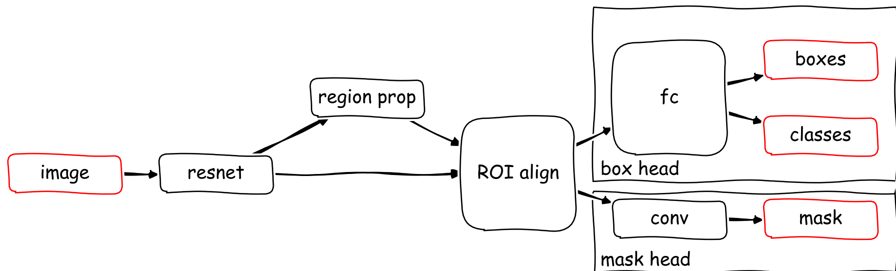
Let's take a model

Mask-RCNN for detection from Facebook AI Research[2]

[2] https://github.com/facebookresearch/maskrcnn-benchmark

Mask-RCNN architecture

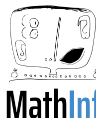
Mask-RCNN from Facebook AI Research[3]

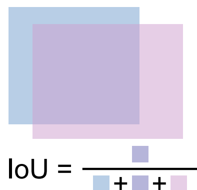


Reasonably complex models, one of the more complex models in vision. What can the JIT do for us?

[3] Original publication: He et al.: Mask R-CNN `https://arxiv.org/abs/1703.06870`
Highly Recommended: Fast AI lecture on Detection:
`https://course.fast.ai/lessons/lesson9.html`

Intersection over Union loss to train the boxes





- Train and target box given by corner x, y and width, height.
- Intersection as max of x, y and min of $x + w$, $y + h$.
- In $[0, 1]$ with $\approx 1 = \text{good}$
- Calculated for many boxes

```
def ratio_iou(x1, y1, w1, h1, x2, y2, w2, h2, eps=1e-5):
    xi = torch.max(x1, x2)                                  # Intersection
    yi = torch.max(y1, y2)
    wi = torch.clamp(torch.min(x1+w1, x2+w2) - xi, min=0)
    hi = torch.clamp(torch.min(y1+h1, y2+h2) - yi, min=0)
    area_i = wi * hi                                        # Area Intersect
    area_u = w1 * h1 + w2 * h2 - wi * hi                    # Area Union
    return area_i / torch.clamp(area_u, min=eps)
```

Why is this not as efficient as it gets?

Easy C++ with custom ops?

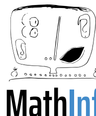
MathInf

```
csrc = """
#include <torch/script.h>

using namespace torch;

Tensor iou_native(const Tensor& x1, const Tensor& y1, const Tensor& w1, con
                  const Tensor& x2, const Tensor& y2, const Tensor& w2, con
    auto xi = torch::max(x1, x2);
    auto yi = torch::max(y1, y2);
    auto wi = torch::clamp(torch::min(x1+w1, x2+w2) - xi, 0);
    auto hi = torch::clamp(torch::min(y1+h1, y2+h2) - yi, 0);
    auto area_i = wi * hi;
    auto area_u = w1 * h1 + w2 * h2 - wi * hi;
    return area_i / torch::clamp(area_u, 1e-5);
}
static auto registry =
  torch::jit::RegisterOperators("super_iou::iou_native", &iou_native);
"""
torch.utils.cpp_extension.load_inline("libsuperiou", csrc,
                                      is_python_module=False, verbose=True)
```

Timing!



I like actual numbers, so let's get some:

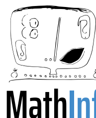
```
  x1, y1, w1, h1, x2, y2, w2, h2 = torch.randn(8, 100, 1000,
                                           device='cuda').exp()
def taketime(fn):
    _ = fn(x1, y1, w1, h1, x2, y2, w2, h2)
    torch.cuda.synchronize() # important!

torch.cuda.synchronize()
%timeit taketime(ratio_iou)
%timeit taketime(torch.ops.super_iou.iou_native)
```

```
1000 loops, best of 3: 1.08 ms per loop
1000 loops, best of 3: 1 ms per loop
```

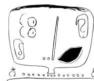
Python overhead 5%-10%, typical for well-vectorized code.

The hard way: Custom kernel



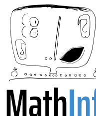

```
template<typename scalar_t>
__global__ void iou_kernel_gpu(PackedTensorAccessor<scalar_t, 1> result,
                              PackedTensorAccessor<scalar_t, 1> x1, ...
                              PackedTensorAccessor<scalar_t, 1> h2
                              ) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i >= x1.size(0)) // we might have more threads than work to do in t
      return;
    // This should look very familiar. We could try reading each element on
    scalar_t xi = max(x1[i], x2[i]);
    scalar_t yi = max(y1[i], y2[i]);
    scalar_t wi = max(min(x1[i]+w1[i], x2[i]+w2[i]) - xi, static_cast<scala
    scalar_t hi = max(min(y1[i]+h1[i], y2[i]+h2[i]) - yi, static_cast<scala
    scalar_t area_i = wi * hi;
    scalar_t area_u = w1[i] * h1[i] + w2[i] * h2[i] - area_i;
    result[i] = area_i / max(area_u, static_cast<scalar_t>(0.00001f));
}
```

The hard way: Custom kernel - glue code




```
torch::Tensor iou_forward(const Tensor& x1, const Tensor& y1, const Tensor&
                          const Tensor& x2, const Tensor& y2, const Tensor&
  auto res = torch::empty_like(x1);
  for (auto& t : {x1, y1, w1, h1, x2, y2, w2, h2}) {
     AT_ASSERTM(t.dim()==1 && t.size(0)==x1.size(0) && t.device()==x1.devic
                "tensors are not of same shape and kind");
  }
  if (x1.is_cuda()) {
    dim3 block(512);
    dim3 grid((x1.size(0)+511)/512);
    AT_DISPATCH_FLOATING_TYPES(x1.type(), "iou", [&] {
      iou_kernel_gpu<scalar_t><<<grid,block>>>(res.packed_accessor<scalar_t
                                  x1.packed_accessor<scalar_t, 1>(), ...);
    });
  } else {
    AT_DISPATCH_FLOATING_TYPES(x1.type(), "iou", [&] {
      iou_kernel_cpu<scalar_t>(res.accessor<scalar_t, 1>(),
                                  x1.accessor<scalar_t, 1>(), ...);
    });
  }
  return res;
```

Custom kernels – too hard!

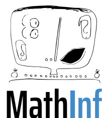


Too hard

- Much more admin things in addition to the algorithm!
- ...and would need CPU code, too.
- And a backward kernel. No autodiff here.
- But it's fast:
  Custom kernel: `10000 loops, best of 3: 86.1` μ`s per loop`
  Pure Python: `1000 loops, best of 3: 1.08 ms per loop`

Can we have fast and easy?

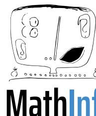
Let the JIT do its thing




```
import math
@torch.jit.script
def ratio_iou_scripted(x1, y1, w1, h1, x2, y2, w2, h2):
    xi = torch.max(x1, x2)                                  # Intersectio
    yi = torch.max(y1, y2)                                  # Intersectio
    wi = torch.clamp(torch.min(x1+w1, x2+w2) - xi, min=0, max=math.inf)
    hi = torch.clamp(torch.min(y1+h1, y2+h2) - yi, min=0, max=math.inf)
    area_i = wi * hi                                      # Area Intersecti
    area_u = w1 * h1 + w2 * h2 - wi * hi     # Area Union
    return area_i / torch.clamp(area_u, min=1e-5, max=math.inf)
```

- Just add `@torch.jit.script`! (and `max` to `clamp`...)
- Much faster than before:
  Custom kernel: 10000 loops, best of 3: 83.4 μs per loop
  Jit: 10000 loops, best of 3: 158 μs per loop
  Pure Python: 1000 loops, best of 3: 1.07 ms per loop
- Relative time closer to custom kernel for larger inputs

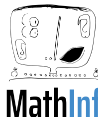
How does it work?



- Slowness comes from storing / reading intermediate results
  Compositionality of NN layers is great, but not always for performance.
- JIT: Python to TorchScript
- JIT Fuser: Find (in particular but not only) pointwise operations and create a custom kernel for them.

```
graph(%x1 : Float(*), ...) {
  %32 : Float(*) = prim::FusionGroup_0(%w2, %h2, %w1, %h1, %y2, %y1, %x2, %
  return (%32);
}
with prim::FusionGroup_0 = graph(%14 : Float(*), ...) {
  %xi : Float(*) = aten::max(%54, %51)
  ...
  %13 : Float(*) = aten::add(%19, %16, %12)
  %8 : int = prim::Constant[value=1]()
  %area_u : Float(*) = aten::sub(%13, %area_i, %8)
  ...
  %2 : Float(*) = aten::div(%area_i, %6)
  return (%2);
```

Automatic backwards



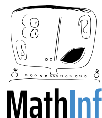
JIT also has automatic backward.
100 loops, best of 3: **5.28 ms** per loop
1000 loops, best of 3: **1.17 ms** per loop

→ 4.5x speedup[4] just by adding `@torch.jit.script`.

Notebook with detailed writeup: *Automatic Optimization with the PyTorch JIT*

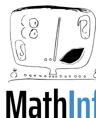
[4] Worked in November, doesn't work with 1.0, will work again soon! – `https://github.com/pytorch/pytorch/pull/14957`

JIT to C++



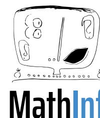

```
#include "torch/script.h"
#include "CImg.h"
using namespace cimg_library;
int main(int argc, char** argv)
{
   CImg<float> image(argv[2]);              // read image
   auto resimg = image.resize(227, 227); // scale to target size
   auto input_ = torch::tensor(torch::ArrayRef<float>(resimg.data(), resimg
   auto input = input_.reshape({1, 3, 227, 227});
   auto = torch::jit::load(argv[1]);              // load model
   std::vector<torch::jit::IValue> inputs;
   inputs.push_back(input);
   auto output = module->forward(inputs).toTensor(); // run model
   auto output_tr = output.clamp(0,255).contiguous(); // show result
   std::cout << output.sizes() << std::endl;
   CImg<float> out_img(output_tr.data<float>(), output_tr.size(2), output_t
   out_img.display("test");
   return 0;
}
```

JIT to C++




```
#include "torch/script.h"
#include "CImg.h"
using namespace cimg_library;
int main(int argc, char** argv)
{
   CImg<float> image(argv[2]);             // read image
   auto resimg = image.resize(227, 227); // scale to target size
   auto input_ = torch::tensor(torch::ArrayRef<float>(resimg.data(), resimg
   auto input = input_.reshape({1, 3, 227, 227});
   auto = torch::jit::load(argv[1]);           // load model
   std::vector<torch::jit::IValue> inputs;
   inputs.push_back(input);
   auto output = module->forward(inputs).toTensor(); // run model
   auto output_tr = output.clamp(0,255).contiguous(); // show result
   std::cout << output.sizes() << std::endl;
   CImg<float> out_img(output_tr.data<float>(), output_tr.size(2), output_t
   out_img.display("test");
   return 0;
}
```

JIT to C++




```
#include "torch/script.h"
#include "CImg.h"
using namespace cimg_library;
```

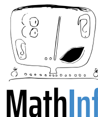
test

```
read image
scale to target size
f<float>(resimg.data(), resimg
7});
 // load model


nsor(); // run model
guous(); // show result

), output_tr.size(2), output_t
```

Tracing the model

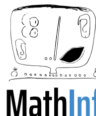


Need to add only three lines to fast neural style example to export model:

```
content_image = content_image[:, :, :227, :227].clone()
traced_script_module = torch.jit.trace(style_model, content_image)
traced_script_module.save("traced-model.pt")
```

Works out of the box!

Tracing complex models



MaskRCNN

- Then bleeding edge, ∼4 person-days (between 1st and 10 November)
- Improve PyTorch for real-world use
    - Allow Tracing of Custom Ops (by Peter G.)
    - Allow Lists of Tensors in Custom Ops
    - Tracing of structures which have dynamic data in Tensors
    - Better error message for file not found

  ... → all in 1.0!

- Needs about 10 changes (∼100 lines)[5] to make the model tracing-friendly,
- some *particularly dynamic* bits implemented using scripting,
- move C++ code from PyTorch extension to custom ops (trivial),
- a way to print labels (implemented using OpenCV as custom op),

[5] `https://github.com/facebookresearch/maskrcnn-benchmark/pull/138`

Android

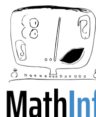
Currently documented way: PyTorch → ONNX → Caffe2[6]

However with Pytorch on Android we would have

- Smoother workflow,
- literally any PyTorch model on Android,
- no bug-prone transfer and conversion,
- can back and forth well between device and development for debugging.

[6] https://pytorch.org/tutorials/advanced/super_resolution_with_caffe2.html

Proof of concept port is not all that hard



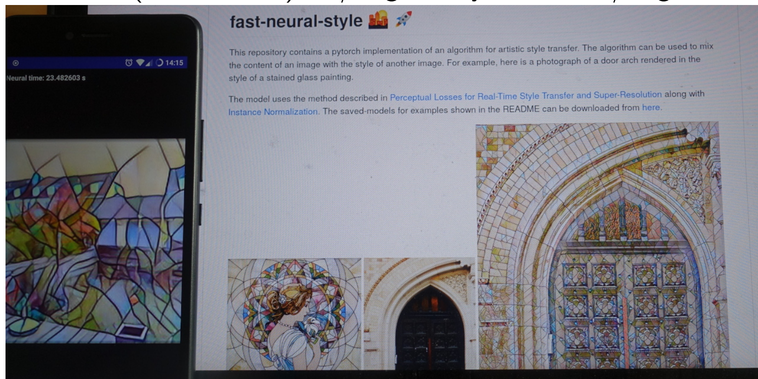
...if you know where to edit things:
NNPACK.cpp mostly from older version of PyTorch, build script copied from "regular" build script (could proably be cleaned up a lot).
Mostly admin stuff to do, not much coding.

```
**/CMakeLists.txt                            |   31 +
aten/src/ATen/Config.h.in                    |    1
aten/src/ATen/core/aten_interned_strings.h |    2
aten/src/ATen/core/interned_strings.h        |    2
aten/src/ATen/native/Convolution.cpp         |   39 +
aten/src/ATen/native/NNPACK.cpp              |  582 ++++++++++++++++++++++++
aten/src/ATen/native/native_functions.yaml |   14
aten/src/TH/THAllocator.cpp                  |   21 -
cmake/**                                     |   56 +-
tools/autograd/derivatives.yaml              |    7
tools/build_pytorch_libs_android.sh          |  444 ++++++++++++++++++++++
```

An easy model on Android: Neural style transfer

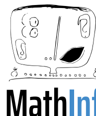
MathInf

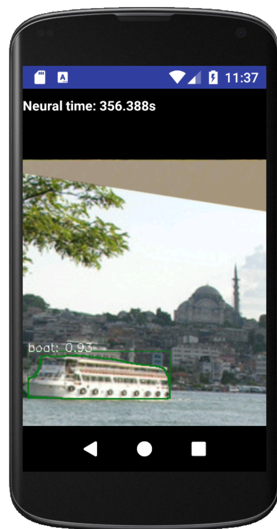
Unchanged Desktop model: 22s/img on my device
Mobilized (32 channels): 2s/img on my device, 1s/img on S8[7]



[7] Simplistic app at `https://lernapparat.de/pytorch-android/`

A complex model on Android: MaskRCNN



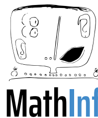


MaskRCNN also works - but it is very slow for now!
Next steps for improvement:

- Mobile-adapted variant of MarkRCNN,
- make some fixed things constants (anchor grid) in MaskRCNN,
- JIT improvements feature: pre-fuse kernels and export those into custom ops,
- Quantization in PyTorch.

Open: How to get those done and also get PyTorch/Android in a good enough shape to publish.

Summary



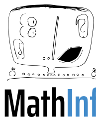
We have seen how to use the PyTorch JIT:

- to help you optimize models with ease,
- export to C++ with tracing (for simple models),
- with tracing + scripting (for more complex models).

Android:

- C++-PyTorch feasible on Android,
- can use arbitrary JIT-exported models directly,
- keeping models in PyTorch (on Python) as long as possible is good for debugging,
- hopefully a thing next year!

If you're interested in these projects, let's have a chat!



Thank you!

Your questions and comments

Contact: Thomas Viehmann, MathInf GmbH, tv@mathinf.eu

Code and slides at

https://lernapparat.de/pytorch-jit-android/