DEEP LEARNING AT SCALE WITH PYTORCH

JEFF SMITH

SENIOR ENGINEERING MANAGER

WHAT IS PYTORCH?

WHAT IS PYTORCH?

EAGER &
GRAPH-BASED
EXECUTION

DYNAMIC
NEURAL
NETWORKS

DISTRIBUTED
TRAINING

HARDWARE
ACCELERATED
INFERENCE

SIMPLICITY
OVER
COMPLEXITY

PYTORCH

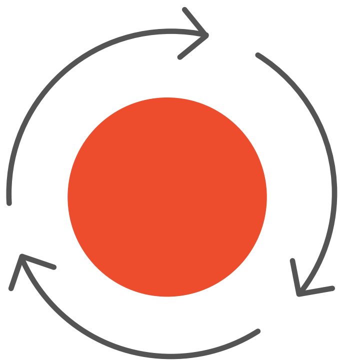
RESEARCH PROTOTYPING + PRODUCTION DEPLOYMENT

CORE PRINCIPLES

CORE PRINCIPLES

DEVELOPER
EFFICIENCY

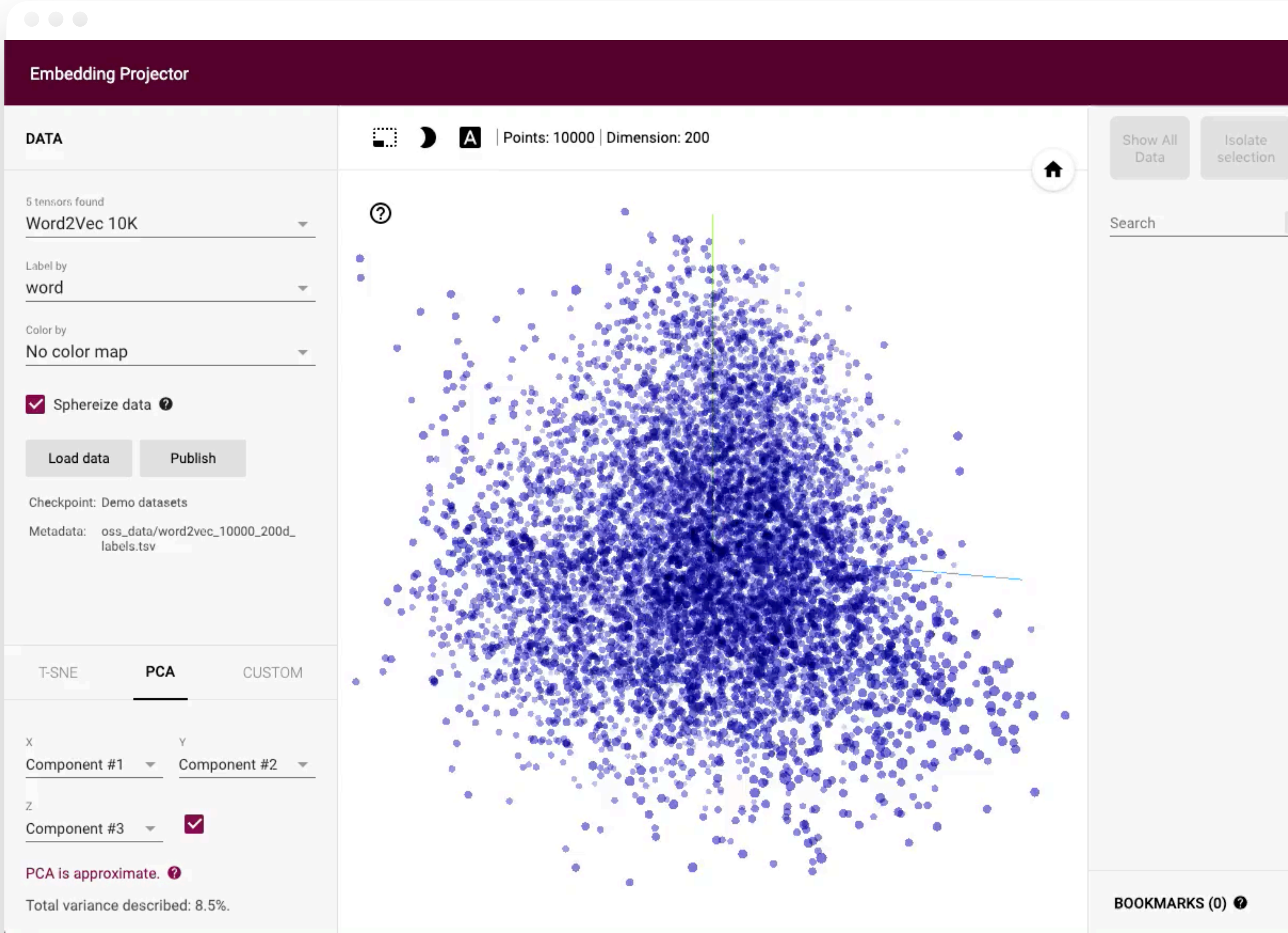
BUILDING
FOR SCALE



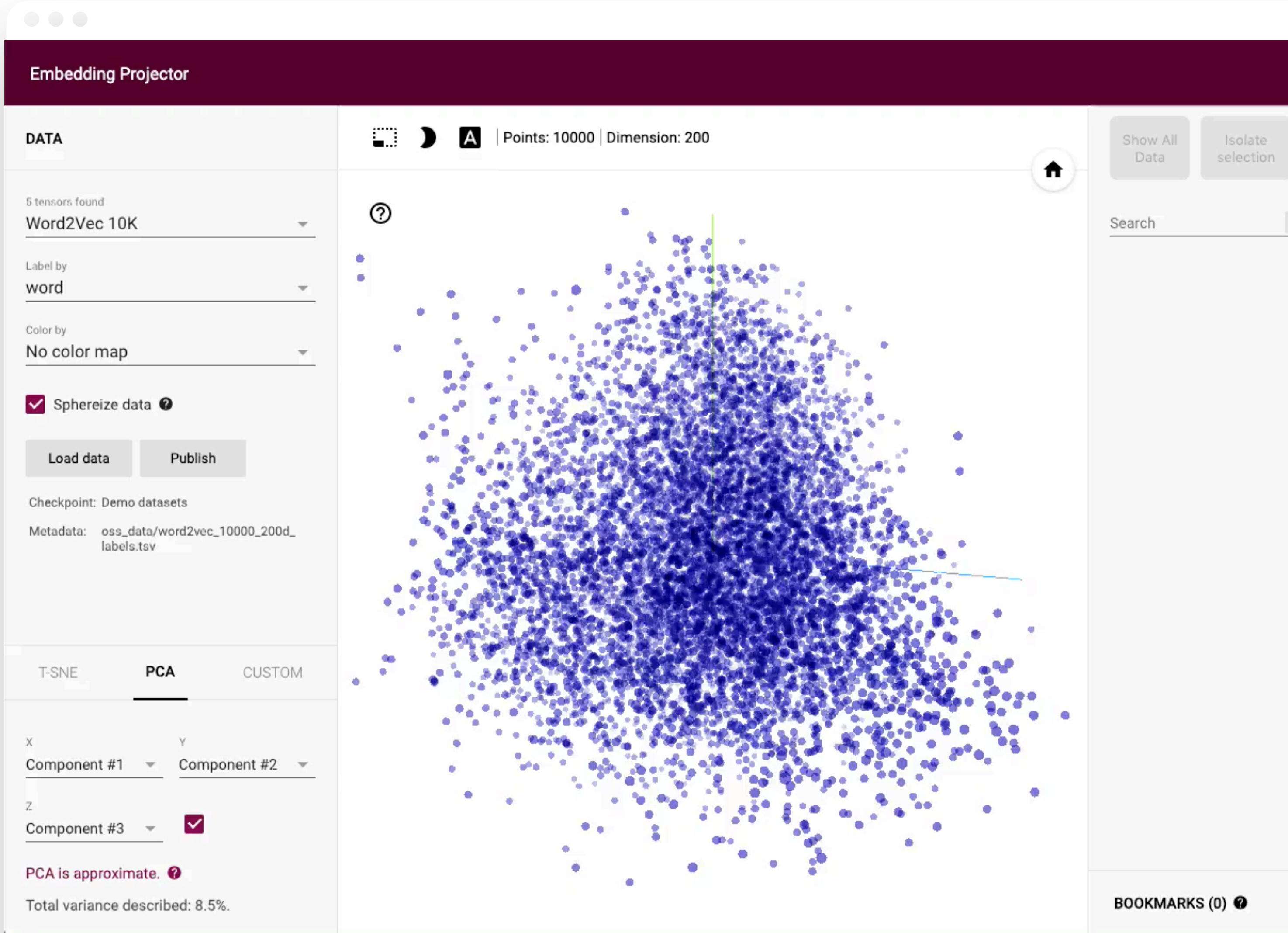
DEVELOPER EFFICIENCY

ENABLING A HIGH VELOCITY OF MODEL ITERATION AND INNOVATION

TENSORBOARD



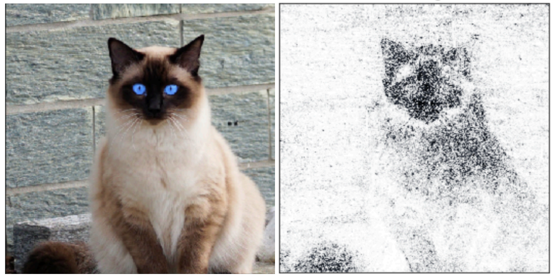
TENSORBOARD



Captum

MODEL INTERPRETABILITY LIBRARY FOR PYTORCH

MULTIMODAL



What color are the cats eyes?

Predicted

Blue (0.517)

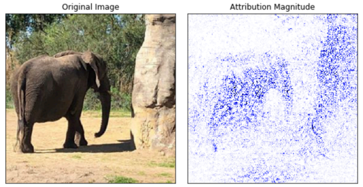
EXTENSIBLE

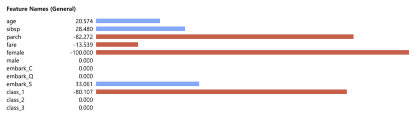
```
class MyAttribution(Attribution):

    def attribute(self, input, ...):
        attributions = self._compute_attrs(input, ... )
        # <Add any logic necessary for attribution>
        return attributions
```

EASY TO USE

visualize_image_attr(attr_algo.attribute(input), ...)



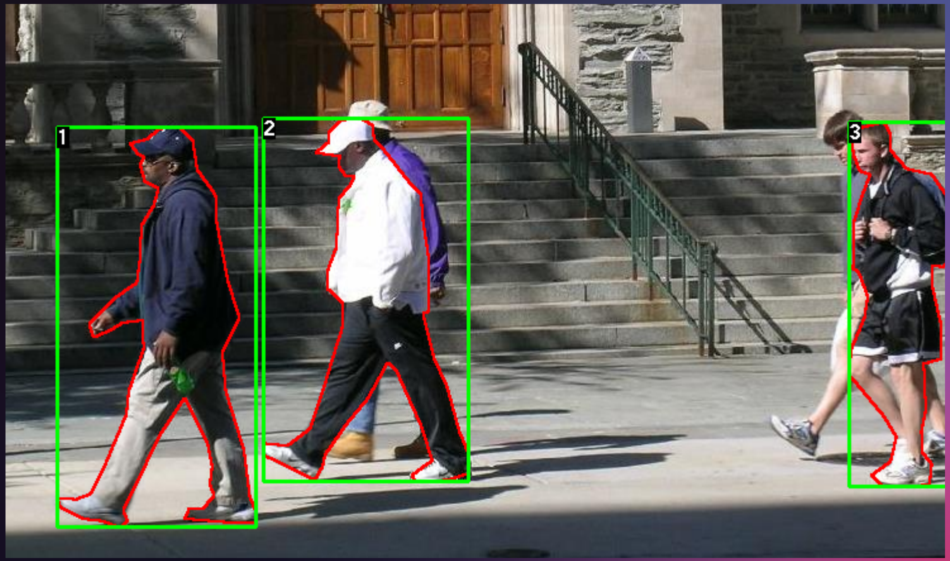


this movie is awful , just awful . someone bought it for me as a christmas present because they knew i liked a good horror flick . i do n't think they understood the " good " part . all i can say is next year this person is getting slipper socks from me . avoid this movie-- it makes you bitter . peace.
<br / >

NAMED TENSORS

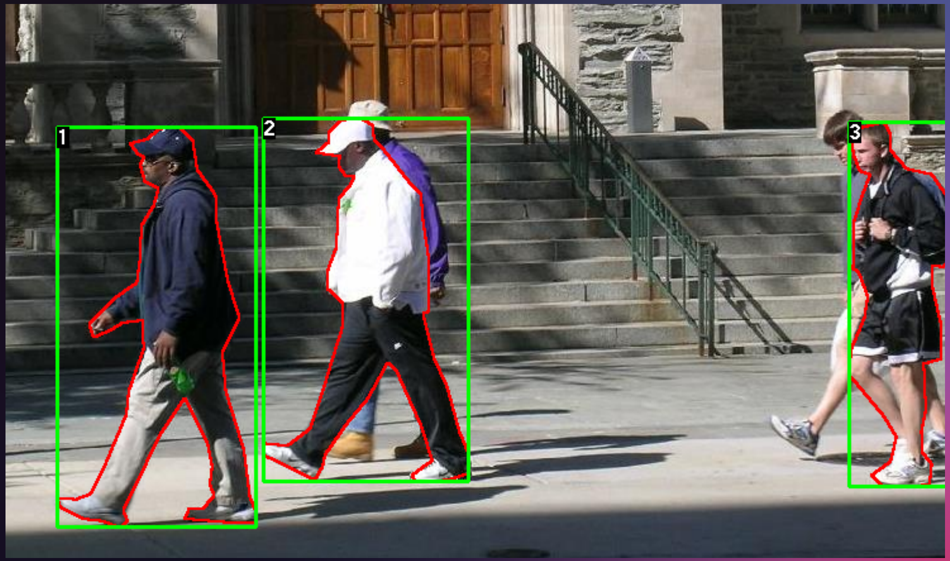
Data has semantic meaning!

But we force users to drop that context and use an abstract "Tensor" mathematical object



Data has semantic meaning!

But we force users to drop that context and use an abstract "Tensor" mathematical object



Key Insight: Named Dimensions

Today we name and access dimensions by comment

```
# Tensor[N, C, H, W]
images = torch.randn(32, 3, 56, 56)
images.sum(dim=1)
images.select(dim=1, index=0)
```

Key Insight: Named Dimensions

Today we name and access dimensions by comment

```
# Tensor[N, C, H, W]
images = torch.randn(32, 3, 56, 56)
images.sum(dim=1)
images.select(dim=1, index=0)
```

But naming explicitly leads to more readable and maintainable code

```
images = torch.randn(N=32, C=3, H=56, W=56)
images.sum('C')
images.select('C', 0)
```

Accidental Alignment

No 1->N broadcast occurs across semantically distinct dimensions, but size happens to match.

```
# image: [N, C, H, W]
# mean:  [C]
# stdv:  [C]
def normalize_image(image, mean, stdv):
    return image.sub_(mean[:, None, None])
                .div_(stdv[:, None, None])
```

Accidental Alignment

No 1->N broadcasting occurs across semantically distinct dimensions, but size happens to match.

```
# image: [N, C, H, W]
# mean:  [C]
# stdv:  [C]
def normalize_image(image, mean, stdv):
    return image.sub_(mean[:, None, None])
                .div_(stdv[:, None, None])
```

But there are so many formats!

```
# PyTorch images: [N, C, H, W]
# PyTorch videos: [N, C, T, H, W]
# TF images:      [N, H, W, C]
# PIL images:     [H, W, C]
```

Accidental Alignment

No 1->N broadcasting occurs across semantically distinct dimensions, but size happens to match.

```
# image: [N, C, H, W]
# mean:  [C]
# stdv:  [C]
def normalize_image(image, mean, stdv):
    return image.sub_(mean[:, None, None])
                .div_(stdv[:, None, None])
```

But there are so many formats!

```
# PyTorch images: [N, C, H, W]
# PyTorch videos: [N, C, T, H, W]
# TF images:      [N, H, W, C]
# PIL images:     [H, W, C]
```

There is a ”time bomb” if I ever normalize the wrong format and the ”unaligned” dimensions have the same size!

Accidental Alignment

No 1->N broadcasting occurs across semantically distinct dimensions, but size happens to match.

```
# image: [N, C, H, W]
# mean:   [C]
# stdv:   [C]
def normalize_image(image, mean, stdv):
    return image.sub_(mean[:, None, None])
                .div_(stdv[:, None, None])
```

Accidental Alignment

No 1->N broadcasting occurs across semantically distinct dimensions, but size happens to match.

```
# image: [N, C, H, W]
# mean:  [C]
# stdv:  [C]
def normalize_image(image, mean, stdv):
    return image.sub_(mean[:, None, None])
                .div_(stdv[:, None, None])
```

If we broadcast by name (*align_as*), we only need a single normalize function for all formats

```
def normalize(t: [..., C, ...],
              mean: [C],
              stdv: [C]):
    return t.sub_(mean.align_as(t))
            .div_(stdv.align_as(t))
```

Named Tensors

Experimental in 1.3

Core Functionality

Common torch operators are supported in eager mode

(Unnamed) autograd is supported

Tutorial

See our in-depth MultiheadedAttention tutorial

Future Work

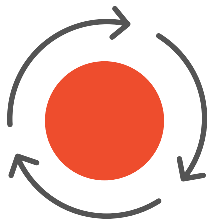
Expanded Coverage

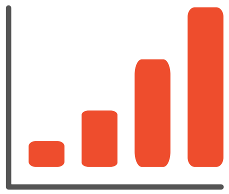
Expanded NN package coverage

Named autograd support

Serialization, multiprocessing, distributed, JIT, mypy

CORE PRINCIPLES

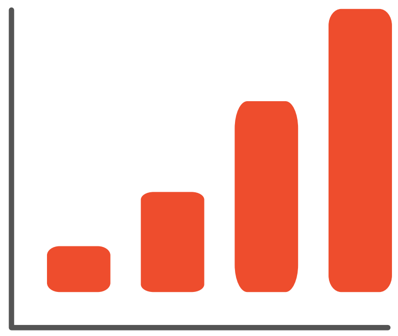




CORE PRINCIPLES

DEVELOPER
EFFICIENCY

BUILDING
FOR SCALE



BUILDING FOR SCALE

HIGH PERFORMANCE EXECUTION FOR MODEL TRAINING AND INFERENCE

OPTIMIZING FOR HARDWARE BACKENDS

OPTIMIZING FOR HARDWARE BACKENDS

PYTORCH DEVELOPMENT ENV

MKL-DNN Cuda/CuDNN
(Q)NNPACK FBGEMM

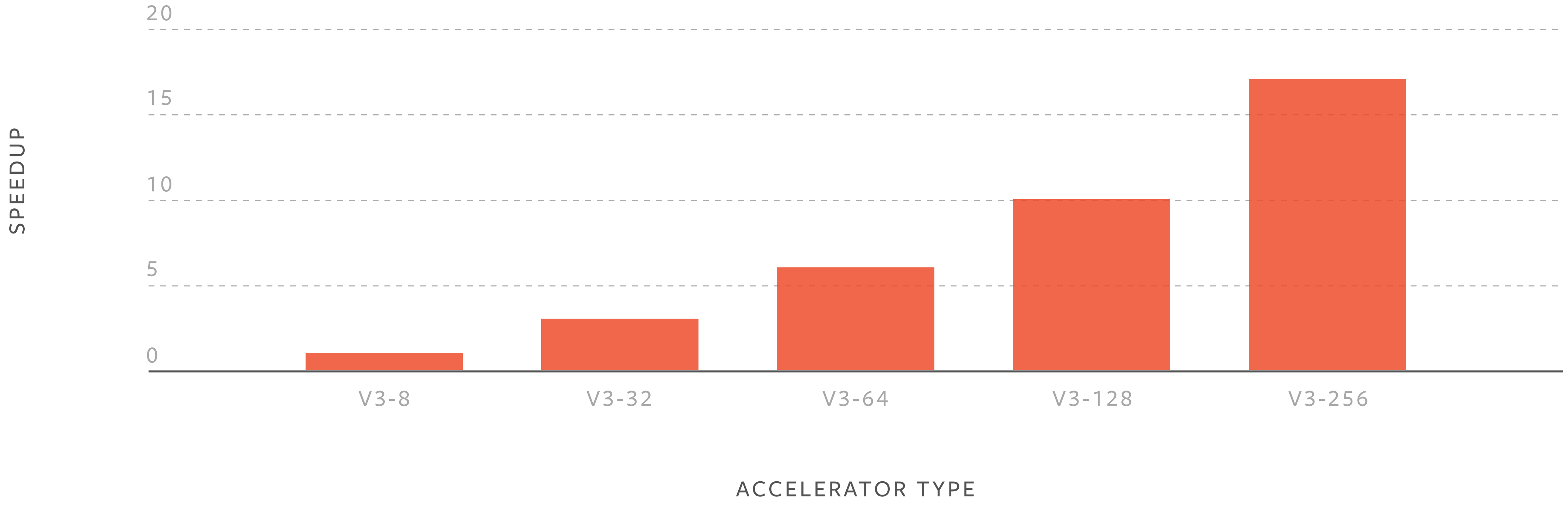
PYTORCH JIT

XLA Glow TVM

CLOUD TPU SUPPORT IN PYTORCH 1.3

github.com/pytorch/xla

TPU PODS SPEEDUP RESNET50 TRAINING WITH IMAGENET DATASET



AVAILABLE NOW:
PYTORCH + CLOUD TPUS IN COLAB

Experiment with PyTorch and Cloud TPUs for free,
right in your browser!

bit.ly/pytorch-tpu

QUANTIZATION

QUANTIZATION

Can neural networks run in lower precision?

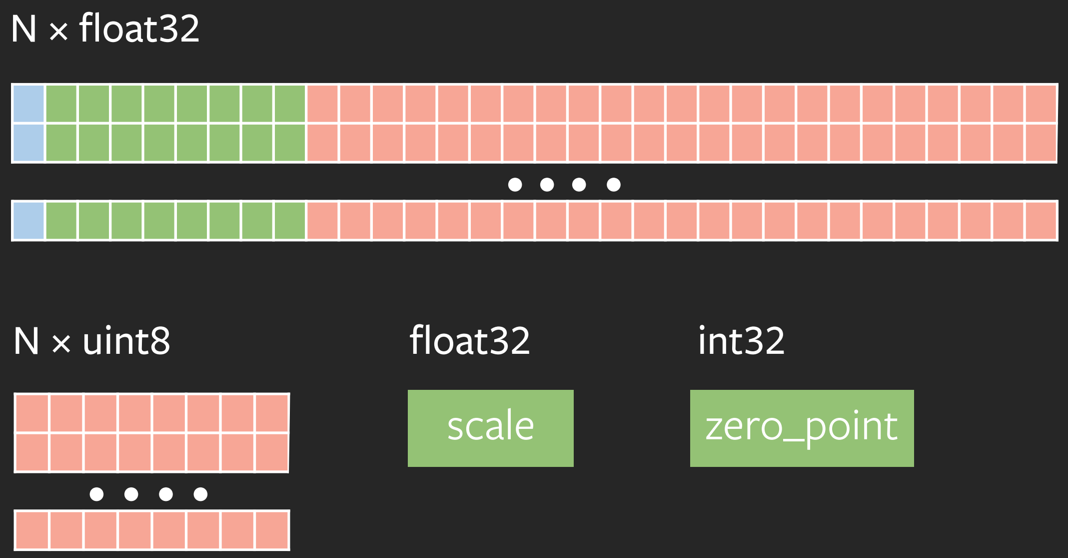
float16, int8

Supported by modern hardware

x86 CPU, ARM CPU, NVidia Volta & Turing, Qualcomm DSP, ...

Maintaining accuracy is hard

Working approaches, ongoing research



float_val = (uint8_val - zero_point) × scale

4x
less memory

2-4x
compute speedup

PYTORCH QUANTIZATION

TURN-KEY WORKFLOWS

Dynamic quantization
Post training quantization
Quantization aware training

COMPONENTS FOR TUNING & RESEARCH

Every part of the workflow is flexible
Use or build your own (in PyTorch)

CORE SUPPORT

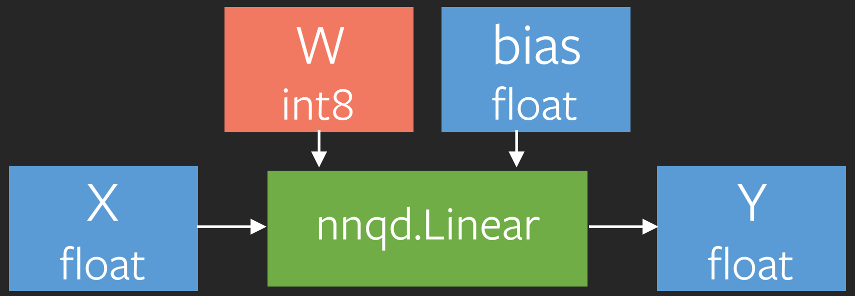
Quantized tensor and operations
Optimized kernels for int8 on x86 and ARM
(other backends coming)

WORKFLOWS

	Quantization	Dataset Requirements	Works Best For	Accuracy
Dynamic Quantization	weights only		small batch LSTMs and MLPs	good
Post Training Quantization	weights and activations	calibration	all	good
Quantization-Aware Training	weights and activations	fine-tuning	all	best

Or build your own

WORKFLOW: DYNAMIC QUANTIZATION



How: one line API

What: quantize weights once, activations at runtime

Good for: LSTMs and MLPs with small batch size

Savings: 2x faster compute, 4x smaller model size

```
# load or train your model
model = WordLanguageModel()
model.load_state_dict(torch.load("model.pt"))

# quantize
qmodel = quantize_dynamic(model,
                         dtype=torch.quint8)

# use or deploy for C++ inference
output = qmodel(input)
torch.jit.script(qmodel).save("scripted.pt")
```

WORKFLOW: POST TRAINING

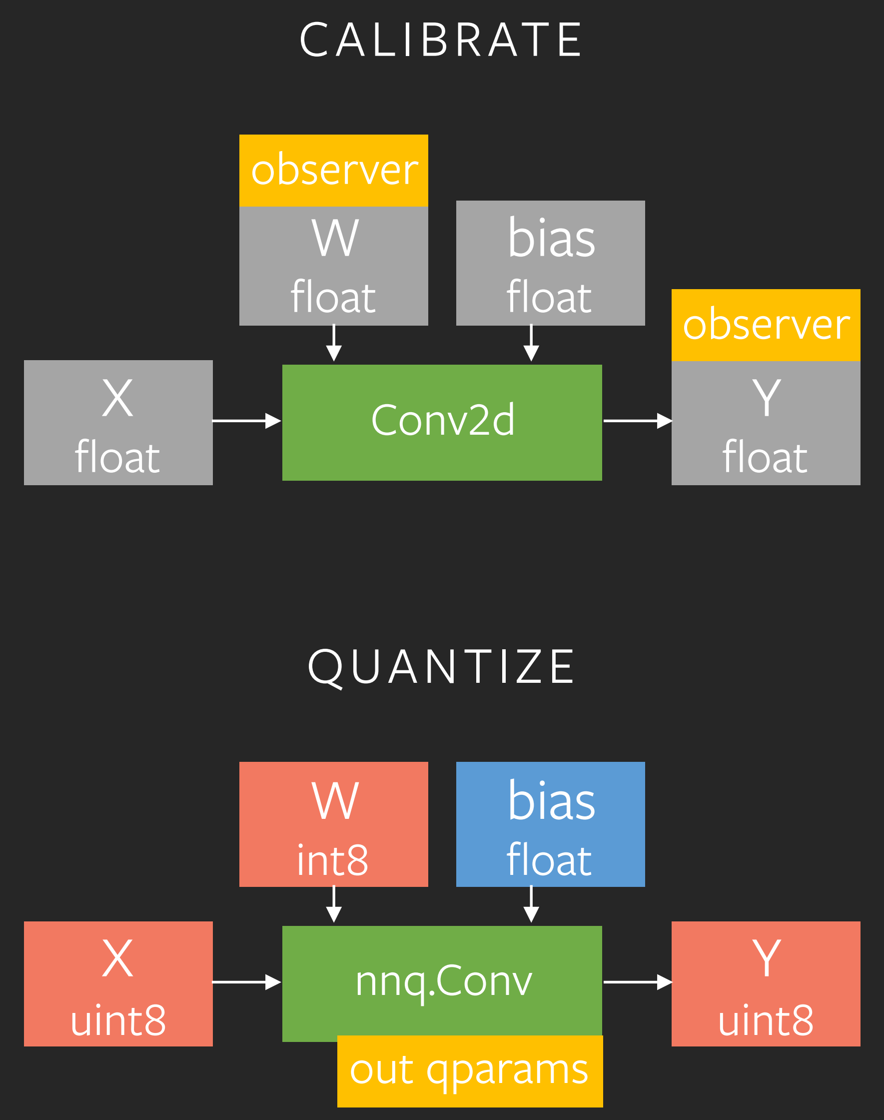
How: tweak model, calibrate on data, convert

What: quantize weight and activations

for entire model or submodules

Good for: CNNs (if the accuracy drop is acceptable)

Savings: 1.5-2x faster compute, 4x less memory



WORKFLOW:
POST TRAINING

How: tweak model, calibrate on data, convert

What: quantize weight and activations

for entire model or submodules

Good for: CNNs (if the accuracy drop is acceptable)

Savings: 1.5-2x faster compute, 4x less memory

```
# load or train your model
model = ResNet50()
model.load_state_dict(torch.load("model.pt"))

# tweak model for best results
# change code directly or use manipulation APIs
model = quantization.fuse_modules(model,
  [["conv1", "bn1", “relu1"]])

print(model.conv1)

  ConvReLU2d(3, 64, kernel_size=(7, 7), … )
```

WORKFLOW: POST TRAINING

How: tweak model, calibrate on data, convert

What: quantize weight and activations

for entire model or submodules

Good for: CNNs (if the accuracy drop is acceptable)

Savings: 1.5-2x faster compute, 4x less memory

```
# load or train your model
model = ResNet50()
model.load_state_dict(torch.load("model.pt"))

# tweak model for best results
# change code directly or use manipulation APIs
model = quantization.fuse_modules(model,
  [["conv1", "bn1", "relu1"]])

# specify which part to quantize and how
qmodel = quantization.prepare(model,
    {"": quantization.default_qconfig})
# configurable!

# collect calibration statistics
qmodel.eval()
for batch, target in data_loader:
        model(batch)

print(model.conv1)

  ConvReLU2d(3, 64, kernel_size=(7, 7), ...
    (observer): MinMaxObserver(
      min_val=0.0, max_val=4.55)
  )
```

WORKFLOW: POST TRAINING

How: tweak model, calibrate on data, convert

What: quantize weight and activations

for entire model or submodules

Good for: CNNs (if the accuracy drop is acceptable)

Savings: 1.5-2x faster compute, 4x less memory

```
qmodel = quantization.prepare(model,
   {"": quantization.default_qconfig})
# configurable!

# collect calibration statistics
qmodel.eval()
for batch, target in data_loader:
        model(batch)

# get the quantized model
qmodel = quantization.convert(qmodel)

print(model.conv1)

  QuantizedConvReLU2d(3, 64,
   scale=0.035, zero_point=0,
   kernel_size=(7, 7), ...)
```

WORKFLOW: POST TRAINING

How: tweak model, calibrate on data, convert

What: quantize weight and activations for entire model or submodules

Good for: CNNs (if the accuracy drop is acceptable)

Savings: 1.5-2x faster compute, 4x less memory

```
# collect calibration statistics
qmodel.eval()
for batch, target in data_loader:
        model(batch)

# get the quantized model
qmodel = quantization.convert(qmodel)

# use or deploy for C++ inference
qmodel(input)
torch.jit.script(qmodel).save("quantized.pt")
```

PYTORCH AT CORE

Same framework, no conversion

- Same serialization
- Python or TorchScript

Eager at its core

- Most logic is in python
- Extensibility, debuggers, stack traces

Extensible API

- New layers
- Observers
- Quantization techniques
- Partial quantization

torch.quantize_per_tensor

torch.quantize_per_channel

torch.nn.quantized.*

torch.nn.quantized.dynamic.*

torch.quantization.*

torch.quantization.Observer

torch.quantization.FakeQuant

EXAMPLE MODELS

	fp32 accuracy	int8 accuracy change	Technique	CPU inference speed up
ResNet50	76.1 Top-1, Imagenet	-0.2 75.9	Post Training	2x 214ms ➙ 102ms, Intel Skylake-DE
MobileNetV2	71.9 Top-1, Imagenet	-0.3 71.6	Quantization-Aware Training	4x 75ms ➙ 18ms OnePlus 5, Snapdragon 835
Translate / FairSeq	32.78 BLEU, IWSLT 2014 de-en	0.0 32.78	Dynamic (weights only)	4x for encoder Intel Skylake-SE

These models and more coming to TorchHub soon

Not STAC Benchmarks

TRY IT NOW

EXPERIMENTAL IN 1.3

QUANTIZATION CORE AND WORKFLOWS

Post training, dynamic and
quantization-aware training
x86 and ARM CPU Backends

Tell us what you think:
GitHub issues
discuss.pytorch.org #quantization

EXAMPLE MODELS

QUANTIZED MODELS AND TUTORIALS TO OBTAIN THEM

ResNet-50
ResNeXt-101
InceptionV3
MobileNetV2
... more to come

COMING IN 1.4

MORE BACKENDS AND JIT WORKFLOW

Simpler workflow for TorchScript
Expanding operator coverage

CORE PRINCIPLES

CORE PRINCIPLES

DEVELOPER
EFFICIENCY

BUILDING
FOR SCALE

PYTORCH
JIT

PRODUCTION REQUIREMENTS

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

TORCHSCRIPT

A static, high-performance subset of Python.

1. Prototype your model with PyTorch
2. Control flow is preserved
3. First-class support for lists, dicts, etc.

```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")
```

PYTORCH JIT

An optimizing just-in-time compiler
for PyTorch programs.

1. Lightweight, thread-safe interpreter
2. Easy to write custom transformations
3. Not just for inference! Autodiff support.

```
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)
```

WHAT'S NEXT?

JIT 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

THANK YOU