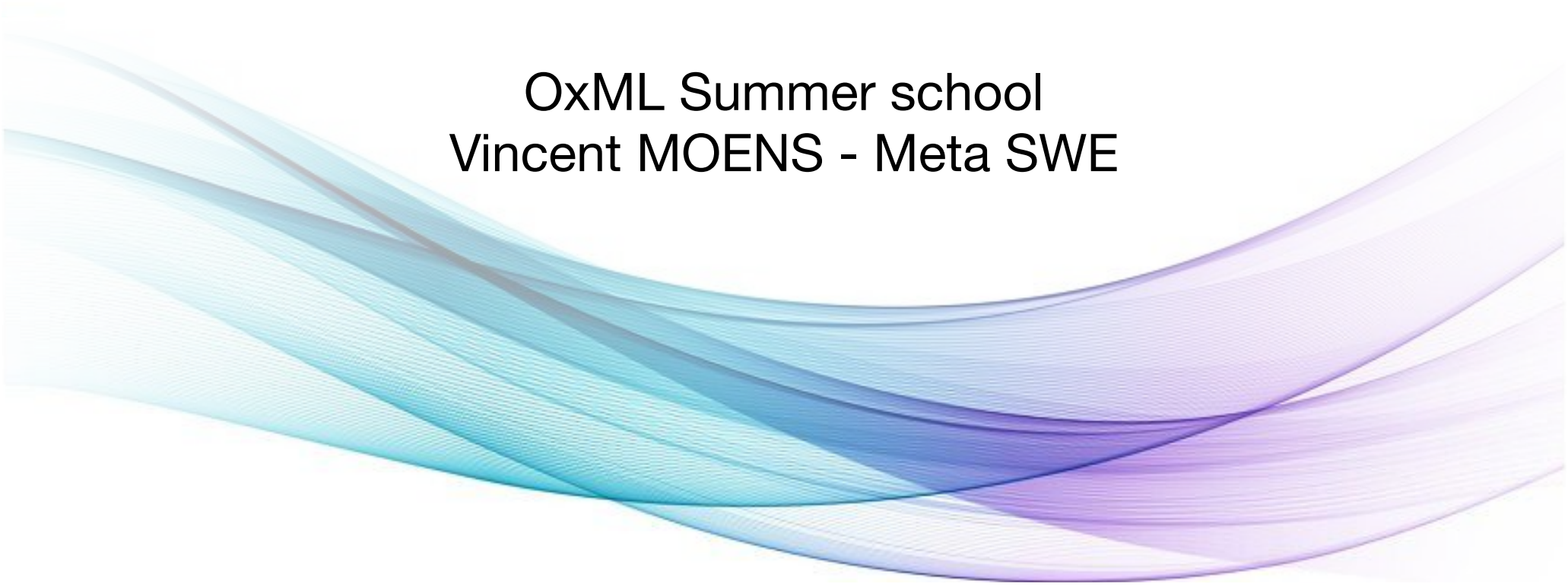
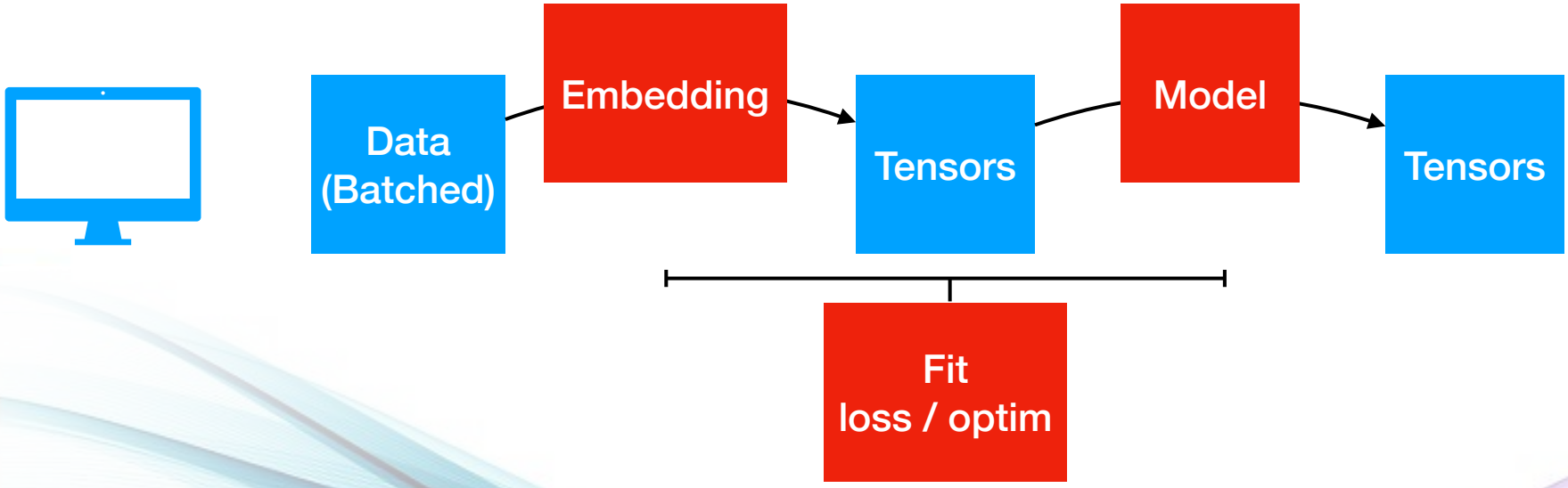
AI frameworks from research to production

OxML Summer school
Vincent MOENS - Meta SWE

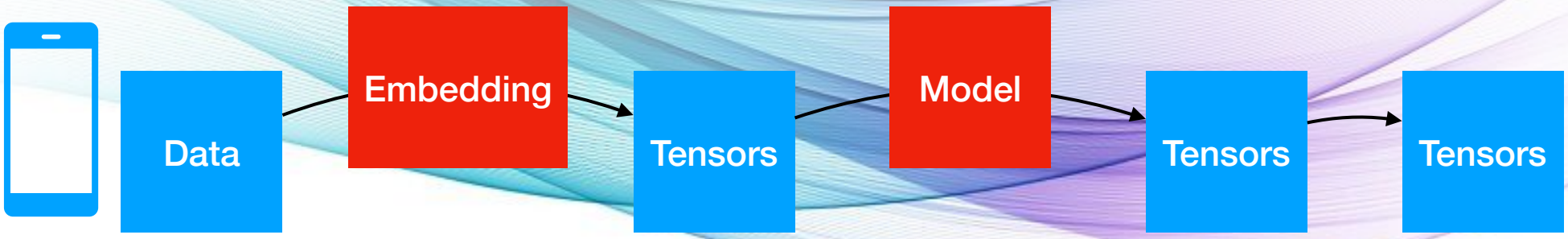


AI Models
From research to production

- Research / training



- Production / execution



AI Models
From research to production

	Research	Production
Bound	Space	CPU
Compiled	-	+
Weird stuff	+++	- -
Data	Static (easy to fetch)	Dynamic
Batched data	++	-
Run on device	- -	+

Weird stuff in research

- Researchers like experimenting
  - Using second order gradients, Jacobians, modifying gradients on-the-fly…
  - For that they’re ready to trade-off some computational speed
  - Usually, one cares first about model performance and then scalability / portability on device (unless the technology is mature)

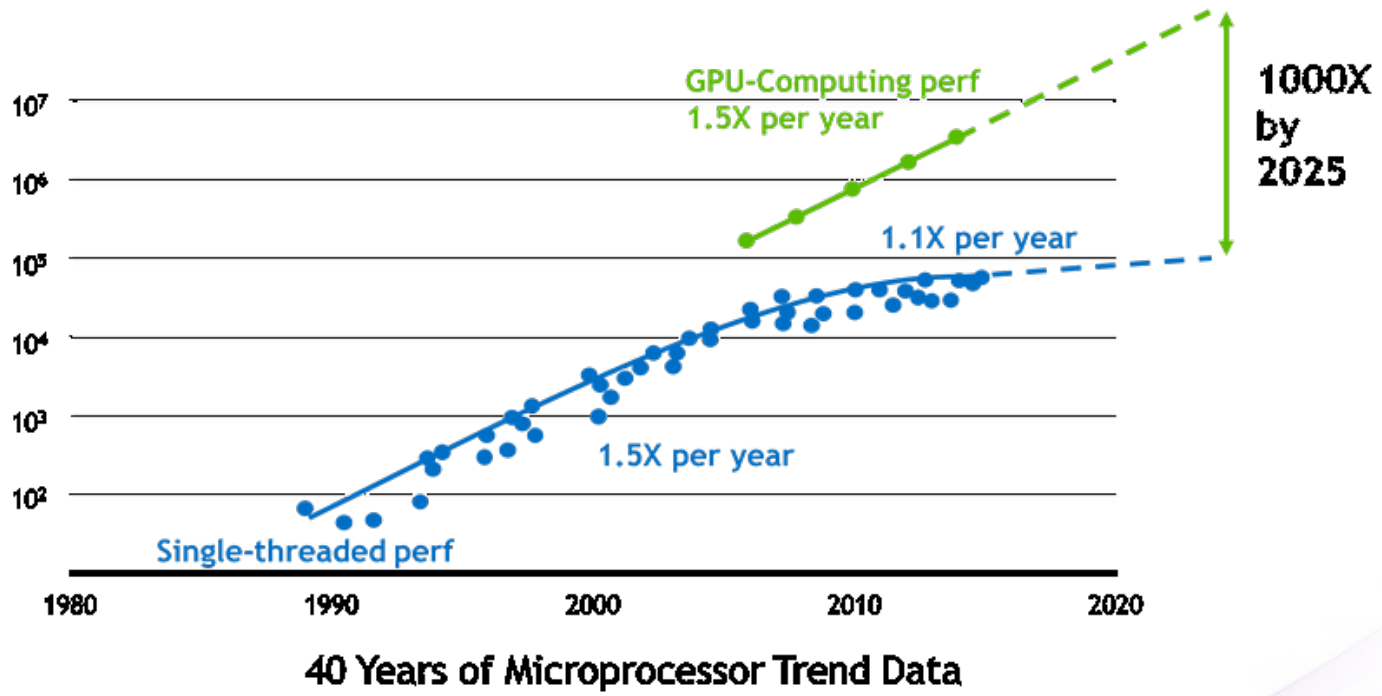
Production

- Execution on server (e.g. data center cooling systems)
- Execution on device (e.g. face recognition)
- Privacy issues: e.g. generative models
- Portability
    - Models should be exportable to a wide variety of environments, from C++ servers to mobile.
- Performance
    - We want to optimize common patterns in neural networks to improve inference latency and throughput.

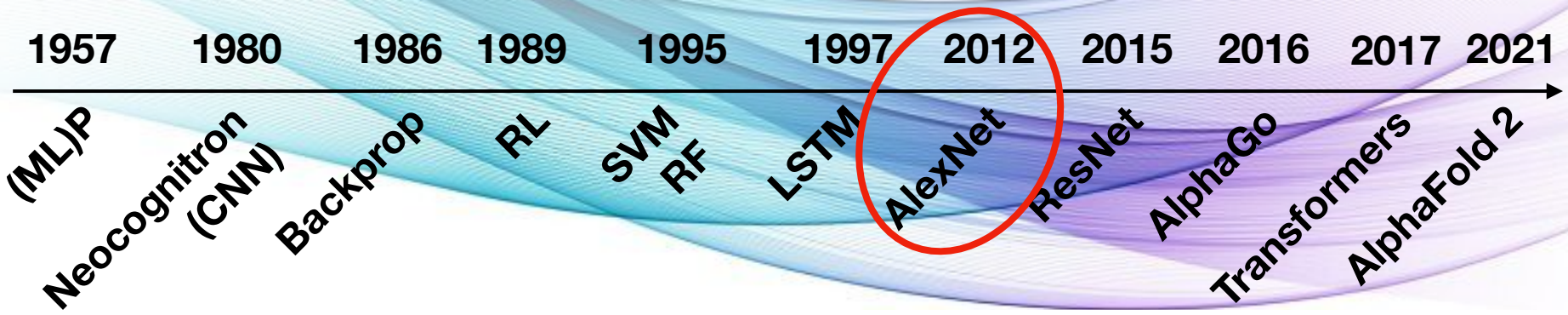
PyTorch

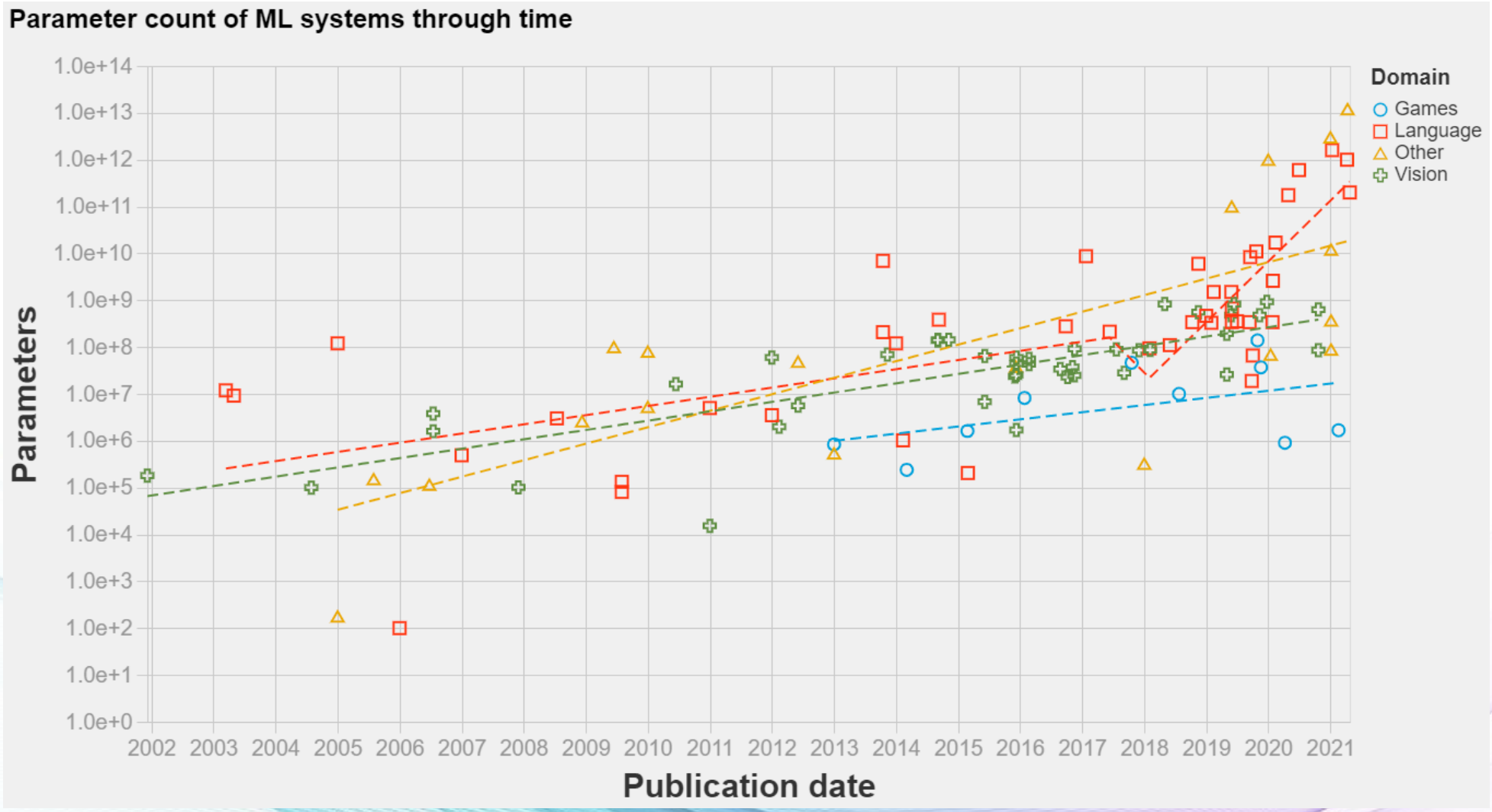
- Initially research-oriented
- Progressively production ready

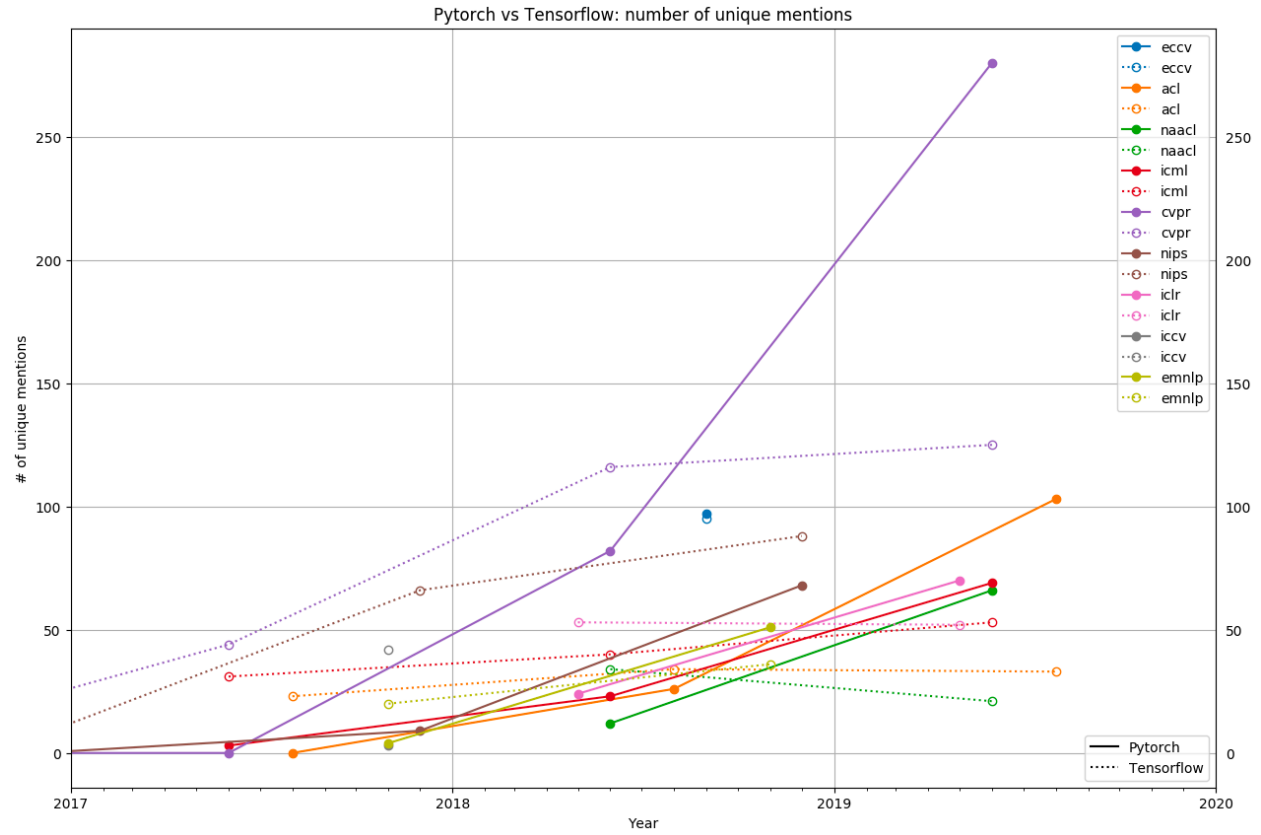
A bit of history

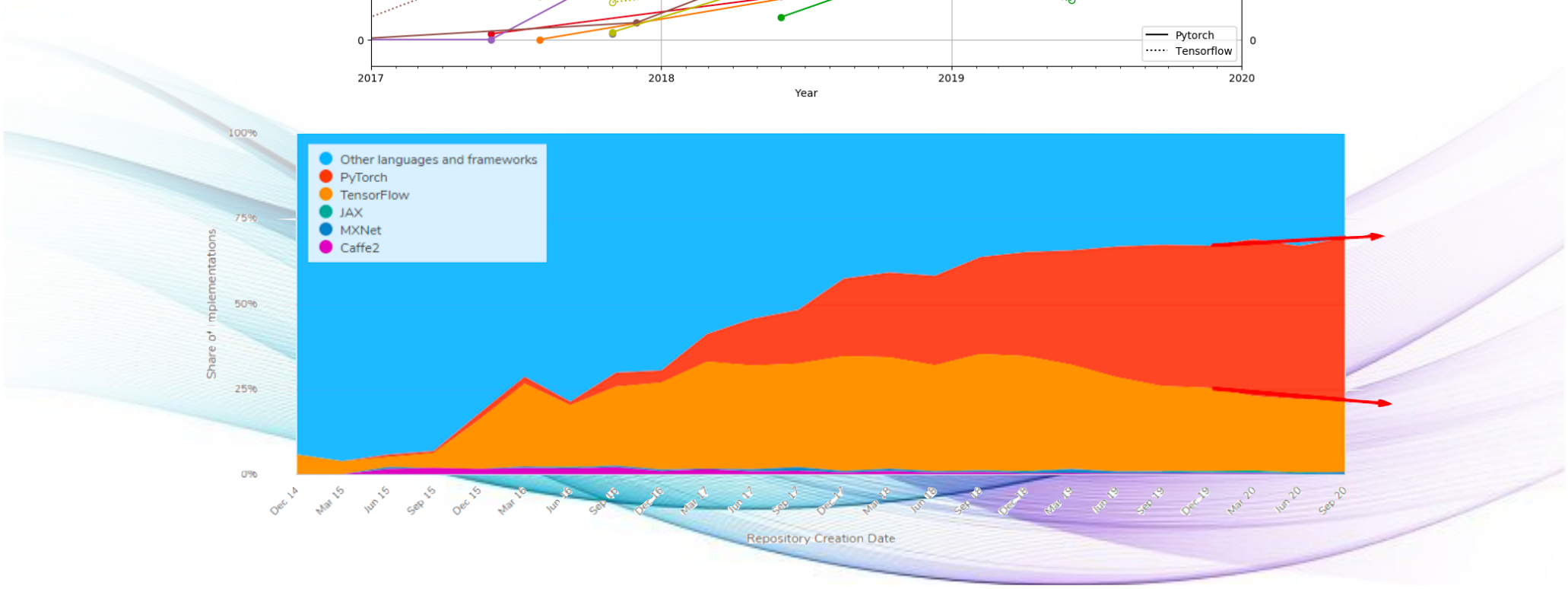


40 Years of Microprocessor Trend Data




Parameter count of ML systems through time
Parameters
1.0e+14
1.0e+13
1.0e+12
1.0e+11
1.0e+10
1.0e+9
1.0e+8
1.0e+7
1.0e+6
1.0e+5
1.0e+4
1.0e+3
1.0e+2
1.0e+1
1.0e+0
2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021
Publication date
Domain
Games
Language
Other
Vision

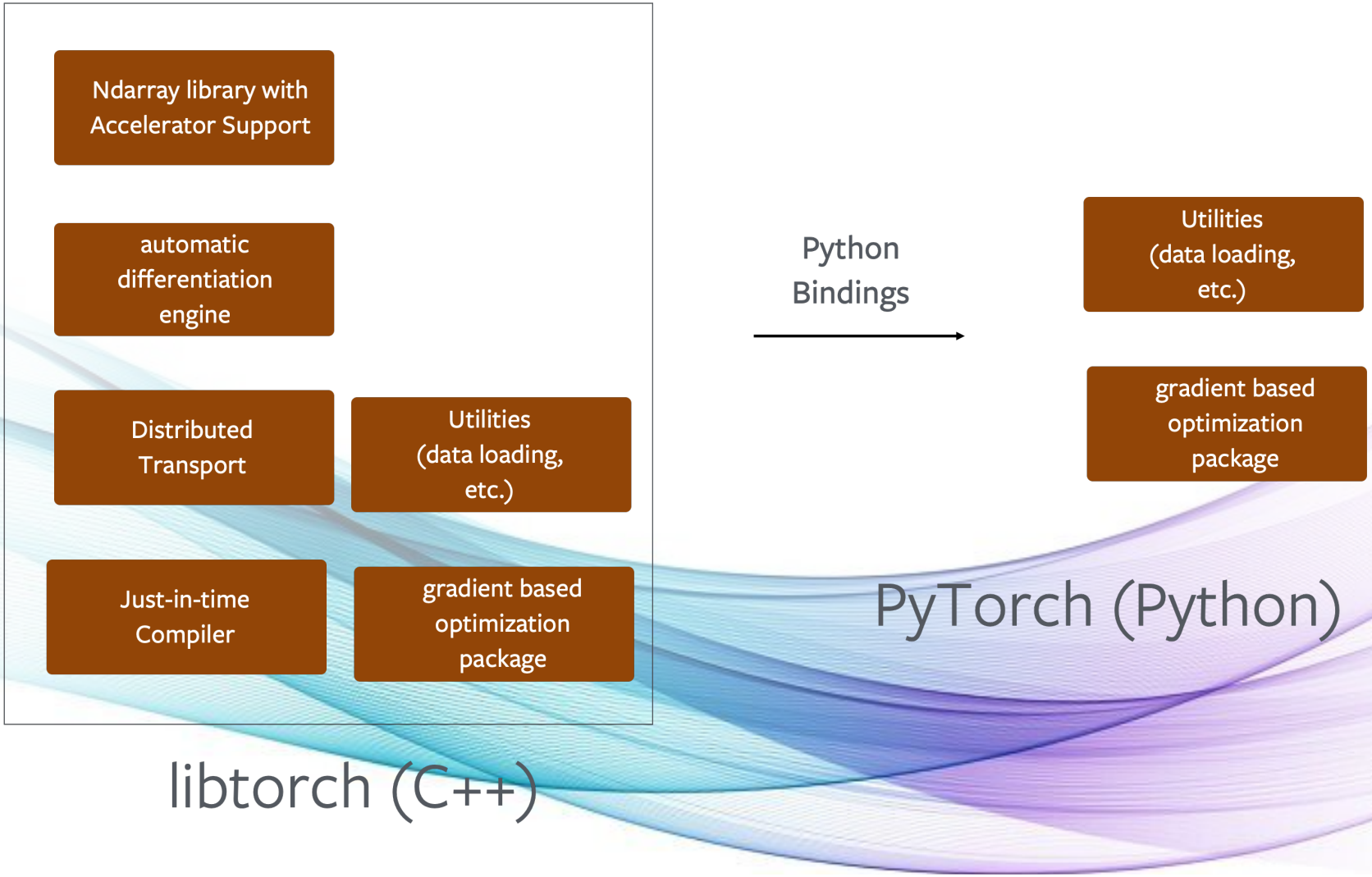

Pytorch vs Tensorflow: number of unique mentions
# of unique mentions
Year
2017
2018
2019
2020
0
50
100
150
200
250
eccv
acl
naacl
icml
cvpr
nips
iclr
iccv
emnlp
Pytorch
Tensorflow


Other languages and frameworks
PyTorch
TensorFlow
JAX
MXNet
Caffe2
Share of Implementations
0%
25%
50%
75%
100%
Repository Creation Date
Dec 14
Mar 15
Jun 15
Sep 15
Dec 15
Mar 16
Jun 16
Sep 16
Dec 16
Mar 17
Jun 17
Sep 17
Dec 17
Mar 18
Jun 18
Sep 18
Dec 18
Mar 19
Jun 19
Sep 19
Dec 19
Mar 20
Jun 20
Sep 20

- Heavily DL oriented
  - Parametric models
  - Exception: GPs/BO, fft, …
- **Torch core**
  - torch.nn
  - torch.autograd
  - torch.distributed
  - torch.optim
  - torch.fx
  - torch.hub
  - torch.linalg
  - functorch (new!)
  - torch.fft
- **Torch ecosystem (incl. domains)**
  - torchvision
  - torchtext
  - torchaudio
  - torchrec (new!)
  - torch multimodal (new!)
  - torchrl (new!)
  - Snapshot (new!): model checkpointing
  - Lightning
  - botorch, gpytorch
  - Torch dynamo (new!)

PyTorch

PyTorch = libtorch + Python bindings



ndarray / Tensor library

- np.ndarray <-> torch.Tensor
- ~450 operations, similar to numpy
- very fast acceleration on GPUs / TPUs / xPUs

```
# let us run this cell only if CUDA is available
# We will use ``torch.device`` objects to move tensors in and out of GPU
if torch.cuda.is_available():
    device = torch.device("cuda")          # a CUDA device object
    y = torch.ones_like(x, device=device)  # directly create a tensor on GPU
    x = x.to(device)                       # or just use strings ``.to("cuda")``
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))       # ``.to`` can also change dtype together!
```

Out:

```
tensor([0.3882], device='cuda:0')
tensor([0.3882], dtype=torch.float64)
```

torch.nn

- torch.nn uses stateful nn.Module instances to create neural nets
- Models can be cast to GPU:
  model.cuda()
- “New”: models can be created on cuda:
  nn.Linear(3, 4, device=0)
- Includes building blocks for MLP, CNN, Transformers, loss modules, activation etc.

```
1   class Net(nn.Module):
2       def __init__(self):
3           super(Net, self).__init__()
4           self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
5           self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
6           self.conv2_drop = nn.Dropout2d()
7           self.fc1 = nn.Linear(320, 50)
8           self.fc2 = nn.Linear(50, 10)
9
10      def forward(self, x):
11          x = F.relu(F.max_pool2d(self.conv1(x), 2))
12          x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
13          x = x.view(-1, 320)
14          x = F.relu(self.fc1(x))
15          x = F.dropout(x, training=self.training)
16          x = self.fc2(x)
17          return F.log_softmax(x)
18
19  model = Net()
20  input = Variable(torch.randn(10, 20))
21  output = model(input)
```

torch.autograd

- torch.autograd:
  - tensor.backward()
  - torch.autograd.grad(y, x)
  - writing your own autograd functions:

```
class MyFun(Function):
    def __forward__(self, x): …
    def __backward__(self, x): …
```

  - forward autodiff (beta)
- AOT autograd << Functorch (see later)

Automatic differentiation

- Forward autodiff
  - Create a dual variable $x(v, j) \in \mathbb{R}^d \times \mathbb{R}^{d \times d}$, initialise $j = I^d$
  - Define a chain of operations $f : \mathbb{R}^m \times \mathbb{R}^{d,m} \to \mathbb{R}^n \times \mathbb{R}^{d,n}$
    Time complexity +
  - Apply the sequence of operations on $x(v, j)$
  - *Best when number of outputs ≥ number of inputs*
- Backward autodiff
  - Create a 'tracked' variable $x \in \mathbb{R}^d$
  - Define a chain of operations $f : \mathbb{R}^m \to \mathbb{R}^n$ and their backward pass $f' : \mathbb{R}^n \to \mathbb{R}^m$
    Space complexity ++
  - Apply the sequence of operations on x to obtain $y \in \mathbb{R}^b$,
    then initialise a gradient vector $g = 1^b$ and backprobagate through the inverse sequence of gradients
  - *Best when number of outputs << number of inputs*

Optimisation

- torch.optim
  - Adam
  - SGD
  - Adamax
  - LBFGS
- Meta-learning: stay tuned!

```
optimizer = optim.SGD(model.parameters(),
     lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2],
     lr=0.0001)

for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

PyTorch Profiling

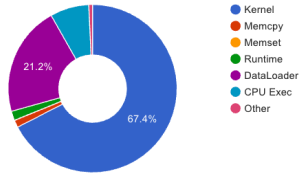
- torch.profiler

Configuration

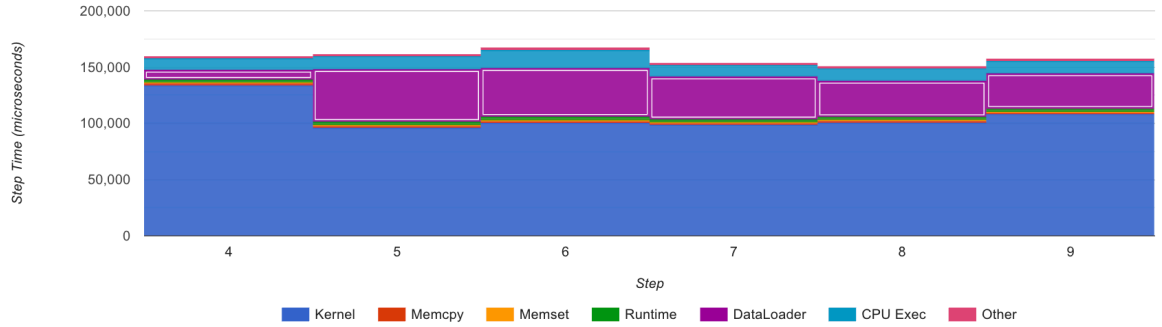
Number of Worker(s)	**1**
Device Type	**GPU**
Number of Device(s)	**1**

Execution Summary

Category	Time Duration (us)	Percentage (%)
Average Step Time	157,532	100
Kernel	106,185	67.41
Memcpy	2,213	1.4
Memset	65	0.04
Runtime	2,816	1.79
DataLoader	33,372	21.18
CPU Exec	11,720	7.44
Other	1,159	0.74



Step Time Breakdown

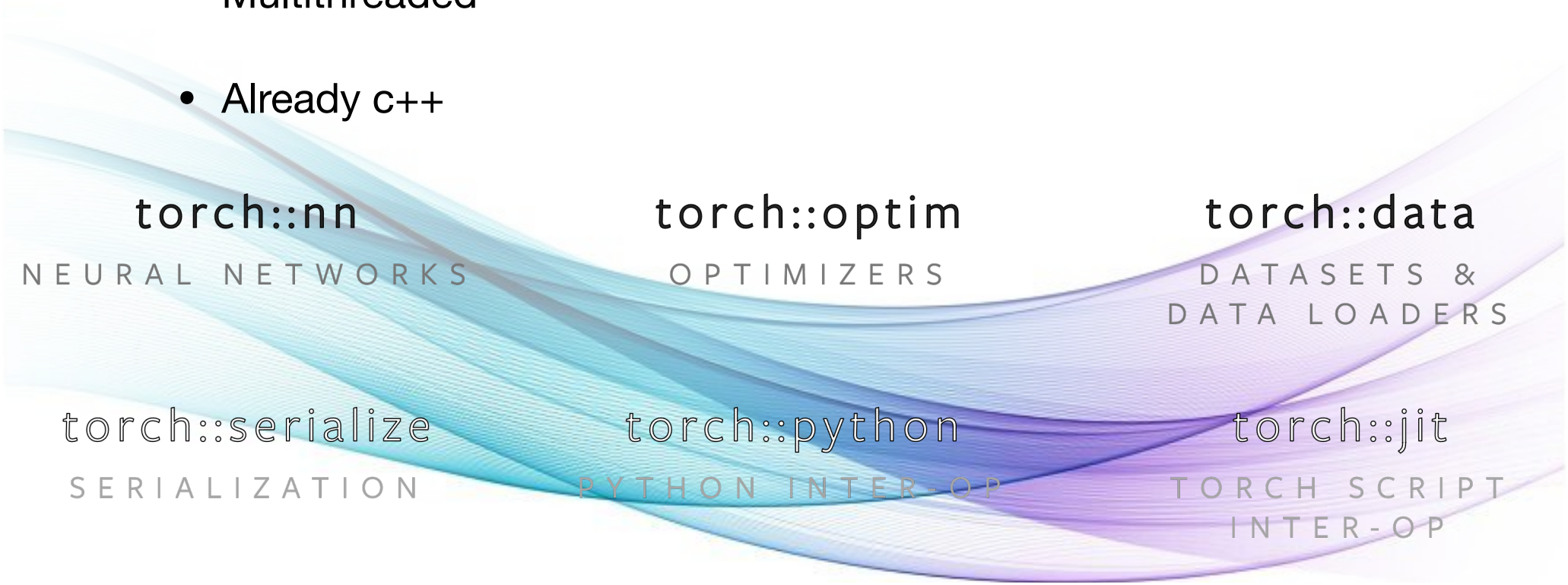


Performance Recommendation

- This run has high time cost on input data loading. 21.2% of the step time is in DataLoader. You could try to set num_workers on DataLoader's construction and enable multi-processes on data loading. Reference: Single- and Multi-process Data Loading

PyTorch in pure C++ frontend (libtorch)

- For research environments that are
  - Low-latency
  - Multithreaded
  - Already c++



Functorch

- Jax-like programming in PyTorch

```
from functorch import vmap
batch_size, feature_size = 3, 5
weights = torch.randn(feature_size, requires_grad=True)

def model(feature_vec):
    # Very simple linear model with activation
    assert feature_vec.dim() == 1
    return feature_vec.dot(weights).relu()

examples = torch.randn(batch_size, feature_size)
result = vmap(model)(examples)
```

```
from functorch import grad
x = torch.randn([])
cos_x = grad(lambda x: torch.sin(x))(x)
assert torch.allclose(cos_x, x.cos())

# Second-order gradients
neg_sin_x = grad(grad(lambda x:
torch.sin(x)))(x)
assert torch.allclose(neg_sin_x, -x.sin())
```

- AOT autograd: compiles the forward and backward ahead of time, 2x speedup on backward
- TorchRL is strongly based on FuncTorch for meta-RL and model ensembling

Domains

- Vision (pytorch/vision)
- Text (pytorch/text)
- Audio (pytorch/audio)
- Multimodal (alpha)
- Graphs (pyg-team)
- RL (alpha)
- BO / GP
- lightning

- Usual pillars:
  - io
  - Models
  - Transforms
  - Ops
  - Datasets

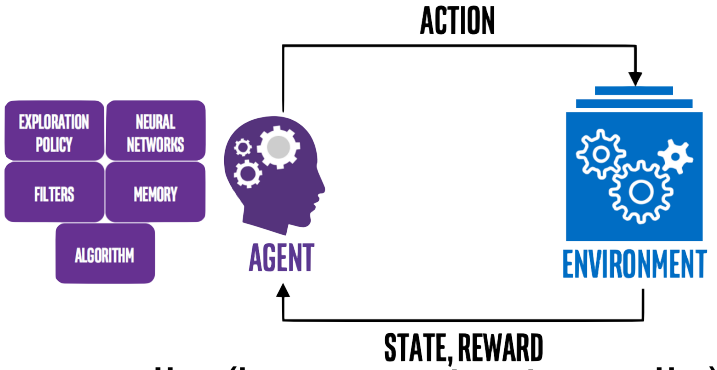
The hard task of a contributor

- Support SOTA models *vs.* Enable tomorrow's research
- Choose what to implement
    - New features can be BC breaking
    - New features will require maintenance.
      What if the engineer leaves? => needs doc
      What if a new, better SOTA solution gets adopted by the community? Users don't like deprecations…
    - Self-contained / limited dependencies
        - New version breaks tests / breaks code
        - Library not maintained anymore: what to do with bugs
- Documentation
- Tests
- Production-friendly

PyTorch Ecosystem example: TorchRL

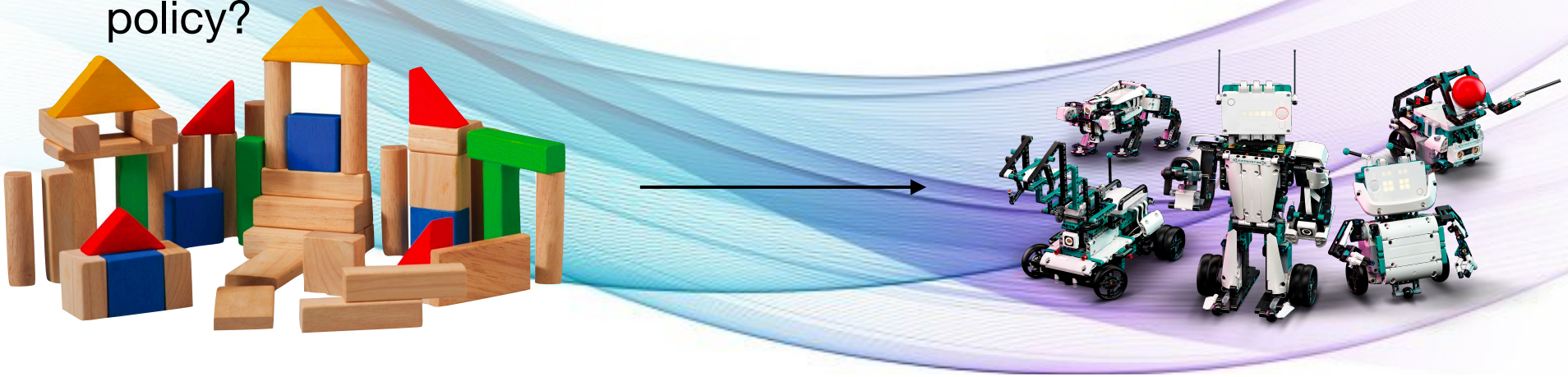
- We want a modular library (*a library, not a framework*)
- e.g. TorchVision:
  - from torchvision.models import ResNet50
    model = ResNet50()
    image = torch.tensor(my_image)
    out = model(image)
  - from torchvision.transforms import ToTensor
    transform = ToTensor()
    image = transform(my_image)
- Much harder for RL!

PyTorch Ecosystem example: TorchRL



- RL is not about the media (images, text, audio) but about the algorithm

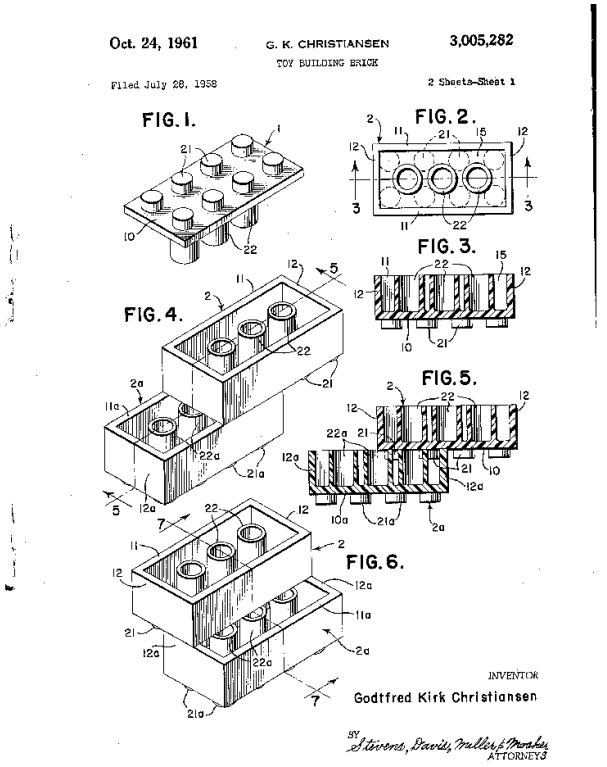
- How does the policy interplay with the environment? How do you collect data? How do you build a loss function that optimizes your policy?



PyTorch Ecosystem example: TorchRL

- RL primitives are unpredictable: a policy can return an action, ± distribution, ± log_prob, ± hidden state, etc.
  An env can return pixels, videos, states, a combination of these.
  Rewards can be scalars or vectors, etc.

- We define a minimum protocol that allows for a generalisation over the space of policies, envs and losses: TensorDict



TorchRL: TensorDict

```
# some initialization

def collector():
    tensordict_data = env.reset()
    while True:
        policy(tensordict_data)
        env.step(tensordict_data)
        yield(tensordict_data)

        tensordict_data = step_tensordict(tensordict_data)

        if some_condition():
            break


for i, tensordict_data in enumerate(collector):
    # store the data in the replay buffer
    replay_buffer.extend(tensordict_data)

    # do some optimization steps
    for j in range(num_optim):
        # sample data from the replay buffer
        optim_data = replay_buffer.sample(batch_size)

        optim_data = optim_data.to(device)

        loss_value = loss_function(optim_data)

        loss_value.backward()
        optimizer.step()
        optimizer.zero_grad()
```

TorchRL: TensorDict

```
a = torch.zeros(3, 4)
b = TensorDict(
    {
    "c": torch.zeros(3, 4, 5, dtype=torch.int32),
    "d": torch.zeros(3, 4, 5, 6, dtype=torch.float32)
    },
    batch_size=[3, 4, 5]
)
tensordict = TensorDict({"a": a, "b": b}, batch_size=[3, 4])
print(tensordict)
```

```
TensorDict(
    fields={
        a: Tensor(torch.Size([3, 4, 1]), dtype=torch.float32),
        b: TensorDict(
            fields={
                c: Tensor(torch.Size([3, 4, 5, 1]), dtype=torch.int32),
                d: Tensor(torch.Size([3, 4, 5, 6]), dtype=torch.float32)},
            batch_size=torch.Size([3, 4]),
            device=cpu,
            is_shared=False)},
    batch_size=torch.Size([3, 4]),
    device=cpu,
    is_shared=False)
```

TorchRL: TensorDict

```
a = torch.zeros(3, 4, 5)
b = torch.zeros(3, 4)
tensordict = TensorDict({"a": a, "b": b}, batch_size=[3, 4])
print(tensordict)
```

```
TensorDict(
    fields={
        a: Tensor(torch.Size([3, 4, 5]), dtype=torch.float32),
        b: Tensor(torch.Size([3, 4, 1]), dtype=torch.float32)},
    batch_size=torch.Size([3, 4]),
    device=cpu,
    is_shared=False)
```

Slicing and indexing

Slicing and indexing is supported along the batch dimensions

```
tensordict[0]
```

```
TensorDict(
    fields={
        a: Tensor(torch.Size([4, 5]), dtype=torch.float32),
        b: Tensor(torch.Size([4, 1]), dtype=torch.float32)},
    batch_size=torch.Size([4]),
    device=cpu,
    is_shared=False)
```

```
tensordict[1:]
```

```
TensorDict(
    fields={
        a: Tensor(torch.Size([2, 4, 5]), dtype=torch.float32),
        b: Tensor(torch.Size([2, 4, 1]), dtype=torch.float32)},
    batch_size=torch.Size([2, 4]),
    device=cpu,
    is_shared=False)
```

```
tensordict[:, 2:]
```

```
TensorDict(
    fields={
        a: Tensor(torch.Size([3, 2, 5]), dtype=torch.float32),
        b: Tensor(torch.Size([3, 2, 1]), dtype=torch.float32)},
    batch_size=torch.Size([3, 2]),
    device=cpu,
    is_shared=False)
```

TorchRL: TensorDict

Stacking

TensorDict supports stacking. By default, stacking is done in a lazy fashion, returning a `LazyStackedTensorDict` item.

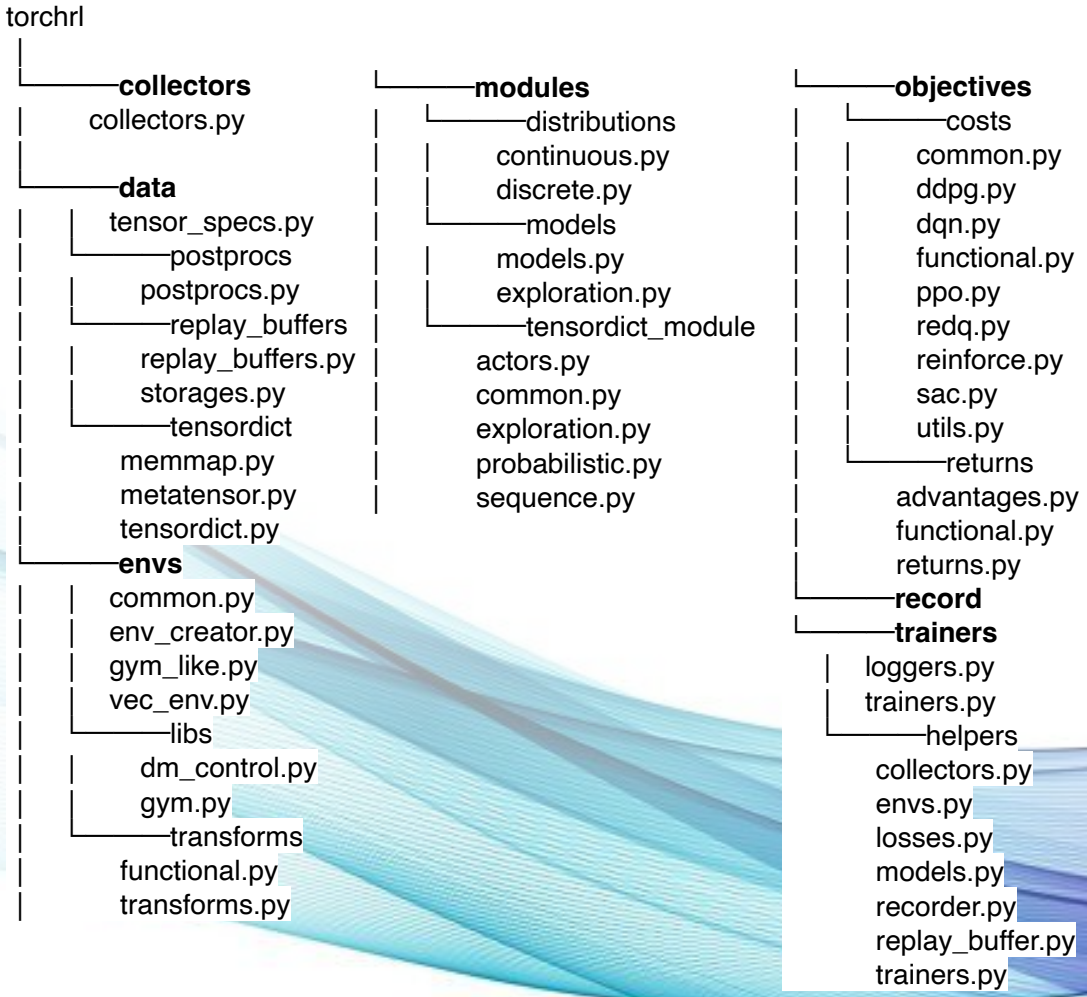
```
# Stack
clonned_tensordict = tensordict.clone()
staked_tensordict = torch.stack([tensordict, clonned_tensordict], dim=0)
print(staked_tensordict)

# indexing a lazy stack returns the original tensordicts
if staked_tensordict[0] is tensordict and staked_tensordict[1] is clonned_tensordict:
    print("every tensordict is awesome!")
```

```
LazyStackedTensorDict(
    fields={
        a: Tensor(torch.Size([2, 3, 4, 5]), dtype=torch.float32),
        b: Tensor(torch.Size([2, 3, 4, 1]), dtype=torch.float32)},
    batch_size=torch.Size([2, 3, 4]),
    device=cpu,
    is_shared=False)
every tensordict is awesome!
```

TorchRL: TensorDict

- TensorDict supports:
  - Indexing (even over very large tensors stored on disk => enables large (>TB) replay buffers)
  - Stacking (easy implementation of rollouts)
  - Casting to device, saving on disk, place in shared memory
  - Unbinding, permuting, reshaping, masking, padding, nesting and de-nesting, etc

PyTorch Ecosystem example: TorchRL



Unlike other libs

- Not a collection of benchmarks
- Properly tested (and documented)
- Follows strict release standards
- Minimal dependency
- Reusable components
- Upstream issues to torch core

Putting code in production

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

```cpp
// C++: load model
auto module = torch::jit::load("serialized_resnet.pt");
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';
```

- Solution for pytorch in production: **torch.jit:** write models in a pythonic way, execute in C++
- Disadvantages:
  - Cumbersome, usually have to made ad-hoc code
  - Limited scope
  - Not necessarily faster

PyTorch 2.0

The Marketing Pitch for PyTorch 2.x

For the average ML Scientist:
30%+ training speedups, lower memory usage
with no changes to code or workflow

For the Compiler/Hardware Engineer
Dramatically easier to write a PyTorch backend

For large-scale DL projects
State of the Art distributed capabilities

For code contributor
Substantially more PyTorch is written in Python

PyTorch 2.0

- PyTorch relies on dynamic graphs:
  - Allows for if/else, random behaviours etc.
  - Graph acquisition and compilation is harder (we can’t know ahead of time how the function will behave)
- Previous solutions:
  - JIT => hardly dynamic, very restrictive and demanding
  - BUT: Makes your code ready for production and mobile (easily ported in C++ from python etc)
  - Performance gain is marginal
- New solution: TorchDynamo
  - Split graph in small, modular sub-graphs
  - No modification of the code is needed

Contributing

- **Supported by companies**
  - Google / Deepmind: TF / Jax
  - Meta: PyTorch
- **Open-sourcing**
  - Mission: support research
    - Pushes us to be forward looking
  - The more users there are, the more likely bugs can be found
  - Window to the company's values and strong position in the AI field
  - External contributors bring value to the products
  - Strong collaborations with academia -> cutting edge solutions

Other resources

- Hugging face
- Paperswithcode
- SageMaker
- Scikit-learn / XGBoost
- scipy

Thanks

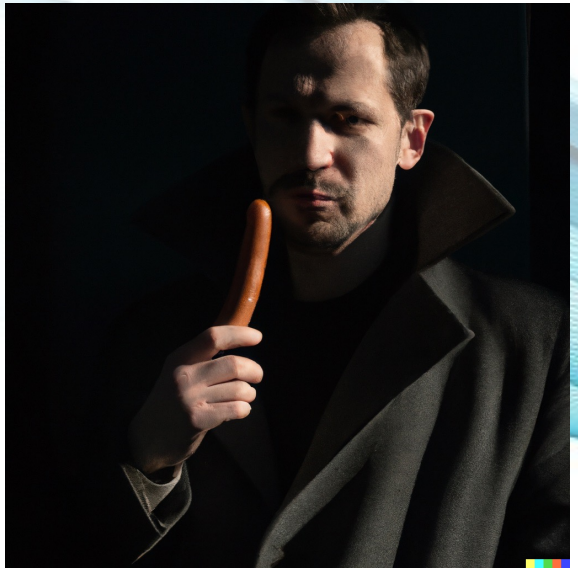
- Feel free to check it, try it and contribute!
  - https://github.com/facebookresearch/rl

Some DALL-E Prompts

Steven, the gnu working in Wall Street as an accountant, manga picture

a pencil drawing of a cargo ship carrying a blue and yellow monster truck

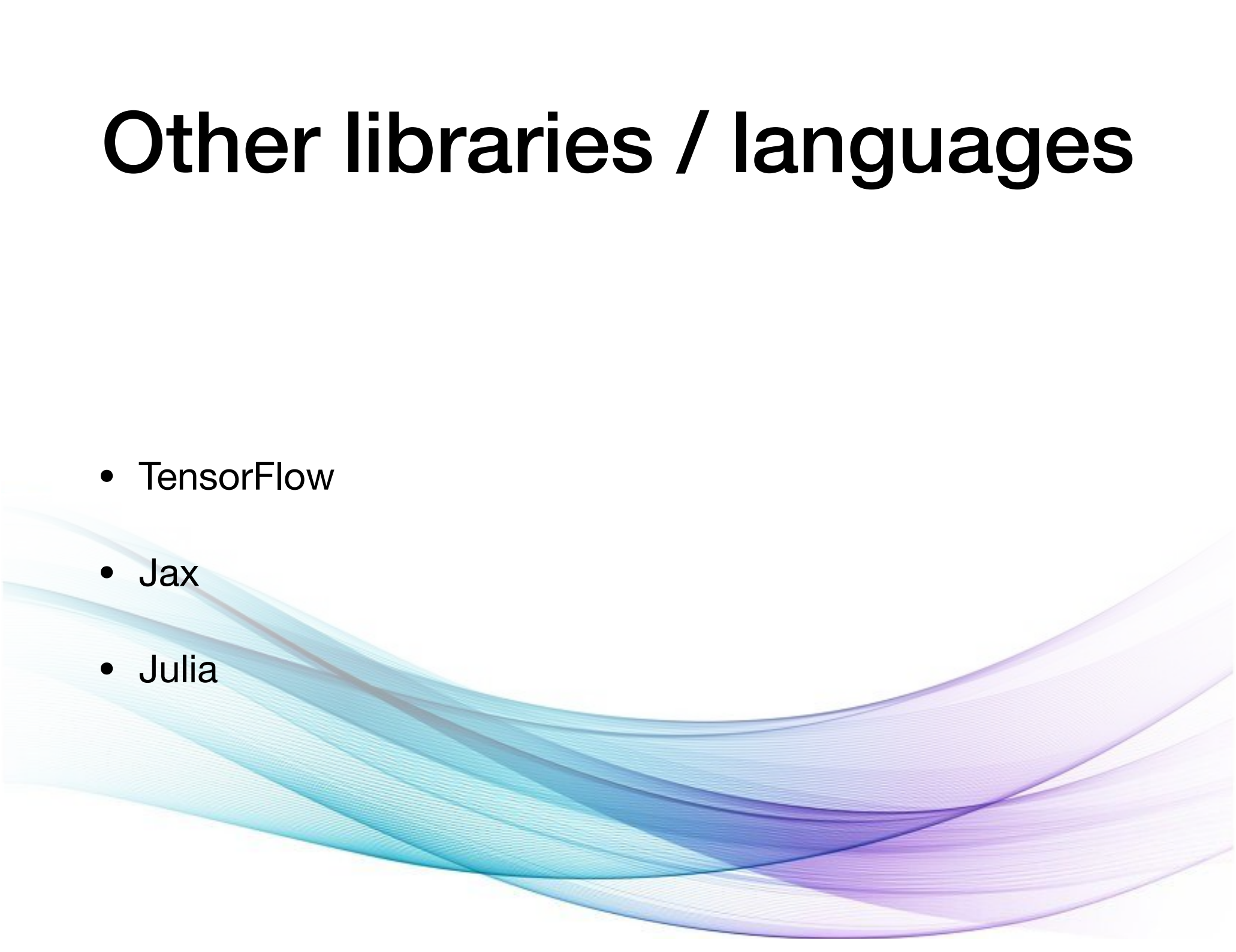
a spy in a dark coat with a sausage in place of a cigar



an ice cream cone made out of rocks, digital art

Other libraries / languages

- TensorFlow
- Jax
- Julia



TensorFlow

- PyTorch + lightning = TF + Keras
- Used to be static (vs dynamic graph)
  - No debugging
- Eager mode since ± v2.0
- XLA:
  - Compiler that greatly accelerates code execution (unlike jit) BUT requires static shapes

Jax

- Autograd + XLA
- “composable transformations of Python+NumPy programs”
- Functional gradients:
  - Allows JVP, VJP, hessian/jacobian computation, etc.

```
def sum_logistic(x):
  return jnp.sum(1.0 / (1.0 + jnp.exp(-x)))

x_small = jnp.arange(3.)
derivative_fn = grad(sum_logistic)
print(derivative_fn(x_small))
```

- Auto-vectorization:
  - Looping over models can be expensive

```
mat = random.normal(key, (150, 100))
batched_x = random.normal(key, (10, 100))

def apply_matrix(v):
  return jnp.dot(mat, v)

@jit
def vmap_batched_apply_matrix(v_batched):
  return vmap(apply_matrix)(v_batched)

print('Auto-vectorized with vmap')
%timeit vmap_batched_apply_matrix(batched_x).block_until_ready()
```