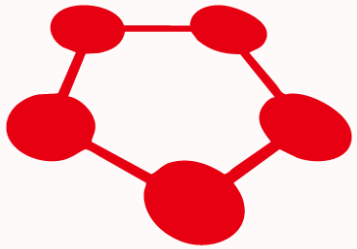
The Frontier of Define-by-Run Deep Learning Frameworks

GTC 2019 @ San Jose. Mar. 20, 2019

Seiya Tokui, Preferred Networks, Inc.

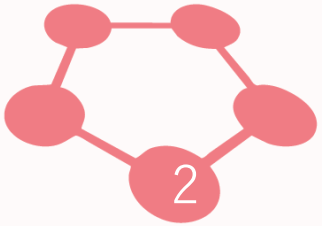
S9380

Chainer

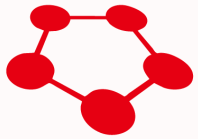


Chainer

Deep Learning Framework
for fast iterative research/development



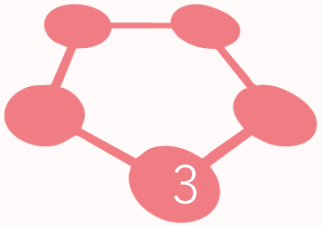
Define-by-Run frameworks



Chainer

PyTorch

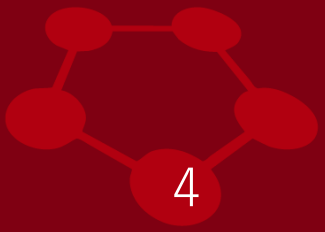
by default from 2.0

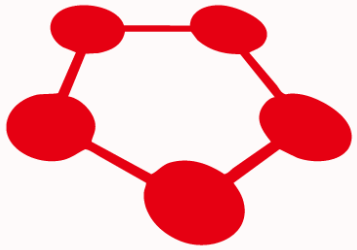


```
x = numpy.array(…)
h1 = layer1(x, W1)
h2 = layer2(h1, W2)
loss = loss_func(h2)

loss.backward()
W1.array -= lr * W1.grad
W2.array -= lr * W2.grad
```

Write forward prop
as a plain Python script.

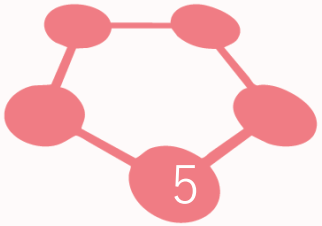
Variables hold how they were computed. Use it to compute the gradient.





Chainer

Deep learning framework
optimized for the **Define-by-Run** API design



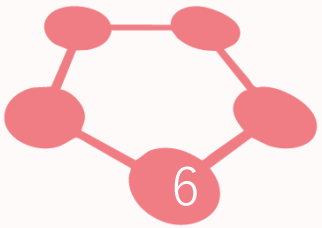
✓ Model description

✓ Distributed training

✓ Serialization, export

……

Everything is optimized for
Define-by-Run style programming

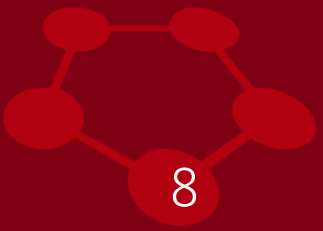


```
class Linear(chainer.Link):

  def __init__(self, n_in, n_out):
    super().__init__()
    with self.init_scope():
      self.W = chainer.Parameter(I.HeNormal(), (n_in, n_out))
      self.b = chainer.Parameter(0, (n_out,))

  def forward(self, x):
    return x @ self.W + self.b
```

Tie parameters to the forward code using OOP.

Object structure =
composition of NN fragments

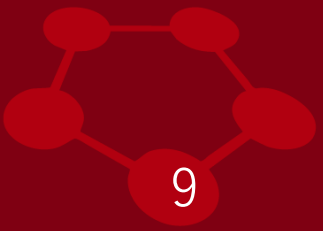
```
class MLP(chainer.Chain):

  def __init__(self):
    super().__init__()
    with self.init_scope():
      self.l1 = Linear(784, 200)
      self.l2 = Linear(200, 100)
      self.l3 = Linear(100, 10)

  def forward(self, x):
    h1 = F.relu(self.l1(x))
    h2 = F.relu(self.l2(h1))
    return self.l3(h2)
```



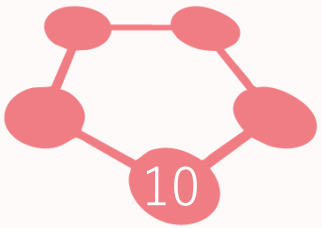
```
for batch in iterator:
  x, t = converter(batch)
  loss = loss_fun(x, t)
  loss.backward()
  optimizer.update()
  model.cleargrad()
```

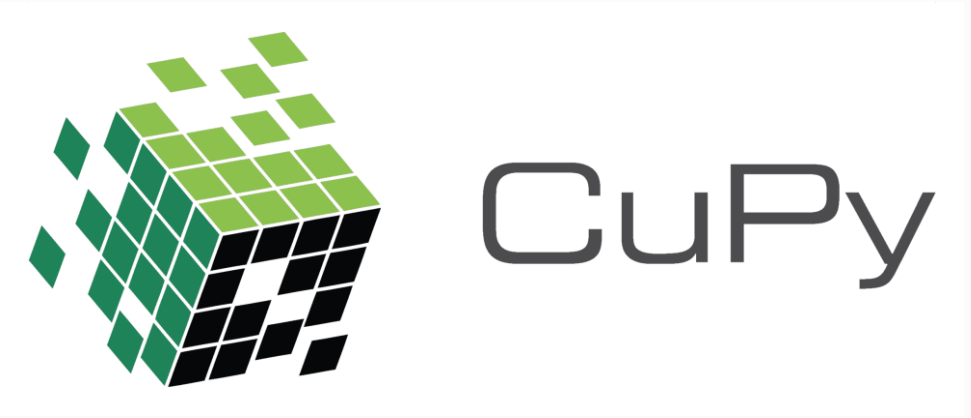
```
# fetch the next minibatch
# concat, transfer to the device
# forward prop
# backprop
# update parameters
# cleanup gradients
```

Every part is plain, customizable Python code

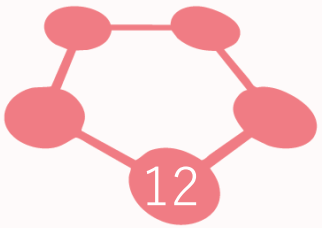


Fast **GPU** computation

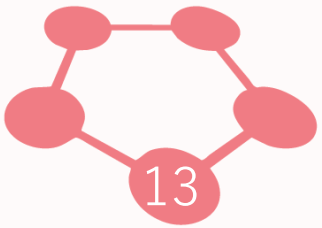



CuPy

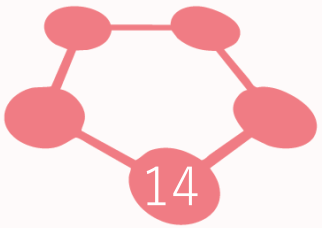
```python
import numpy as np

def logsumexp(x):
    x_max = x.max(axis=1, keepaxis=True)
    x0 = x – x_max
    lse = np.log(np.exp(x0).sum(axis=1))
    lse += x_max
    return lse

x = np.array([...], dtype=np.float32)
print(logsumexp(x))
```



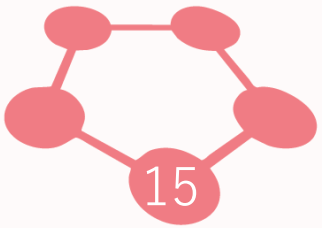
```
import cupy as cp

def logsumexp(x):
    x_max = x.max(axis=1, keepaxis=True)
    x0 = x – x_max
    lse = cp.log(cp.exp(x0).sum(axis=1))
    lse += x_max
    return lse

x = cp.array([...], dtype=np.float32)
print(logsumexp(x))
```

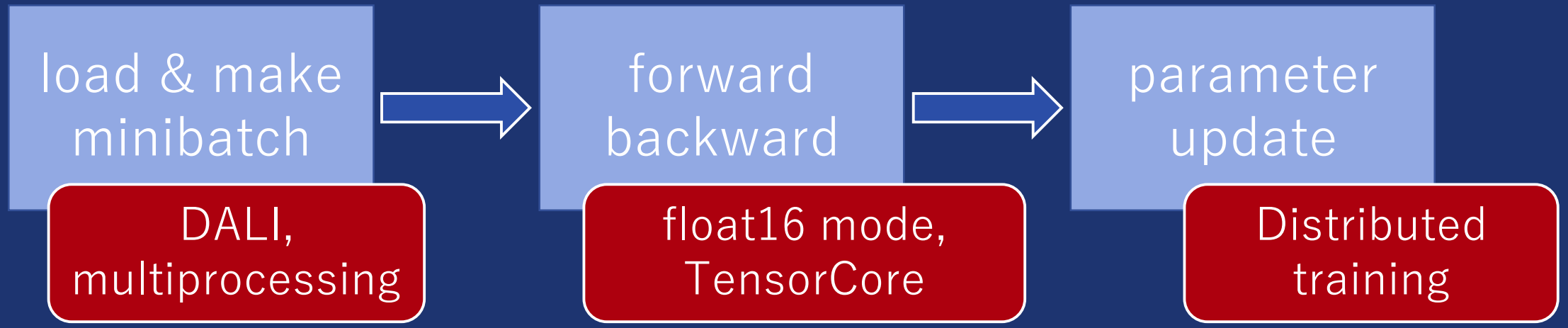


```
import cupy as cp, numpy as np

def logsumexp(x):
    x_max = x.max(axis=1, keepaxis=True)
    x0 = x – x_max
    lse = np.log(np.exp(x0).sum(axis=1))
    lse += x_max
    return lse

x = cp.array([...], dtype=np.float32)
print(logsumexp(x))
```



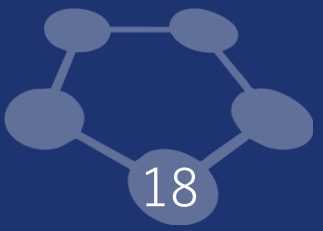
- ✓ cuDNN support (conv, pooling, LSTM, …)
- ✓ Easy custom kernel compiled at runtime
- ✓ FP16 support



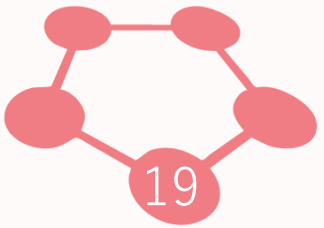

load & make minibatch
forward backward
parameter update
DALI, multiprocessing
float16 mode, TensorCore
Distributed training

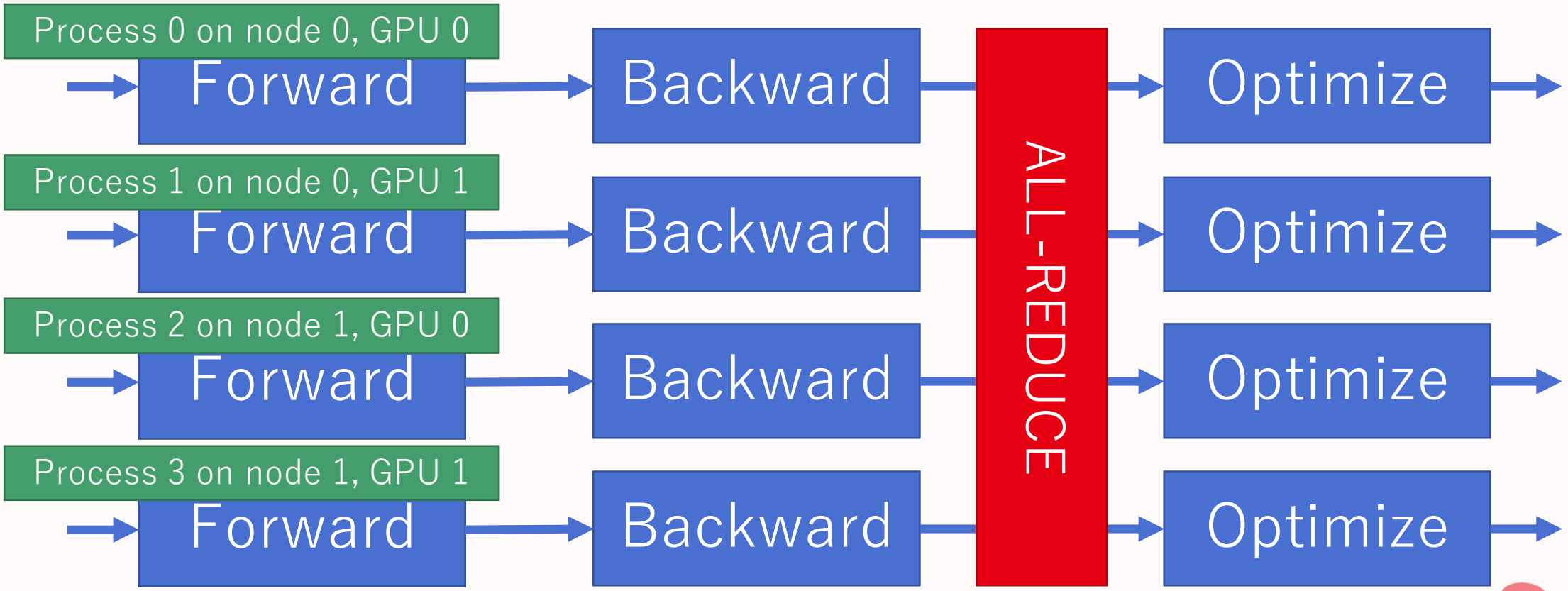
Mixed precision training

- **TensorCore support**
  automatically available
- **Techniques for mixed precision training**
  optimizer.set_loss_scale(scale)
  optimizer.use_fp32_update()
- **mixed16 mode** (coming soon)
  CHAINER_DTYPE=mixed16



Distributed training




Process 0 on node 0, GPU 0
Forward
Backward
ALL-REDUCE
Optimize
Process 1 on node 0, GPU 1
Forward
Backward
Optimize
Process 2 on node 1, GPU 0
Forward
Backward
Optimize
Process 3 on node 1, GPU 1
Forward
Backward
Optimize

Data parallelism

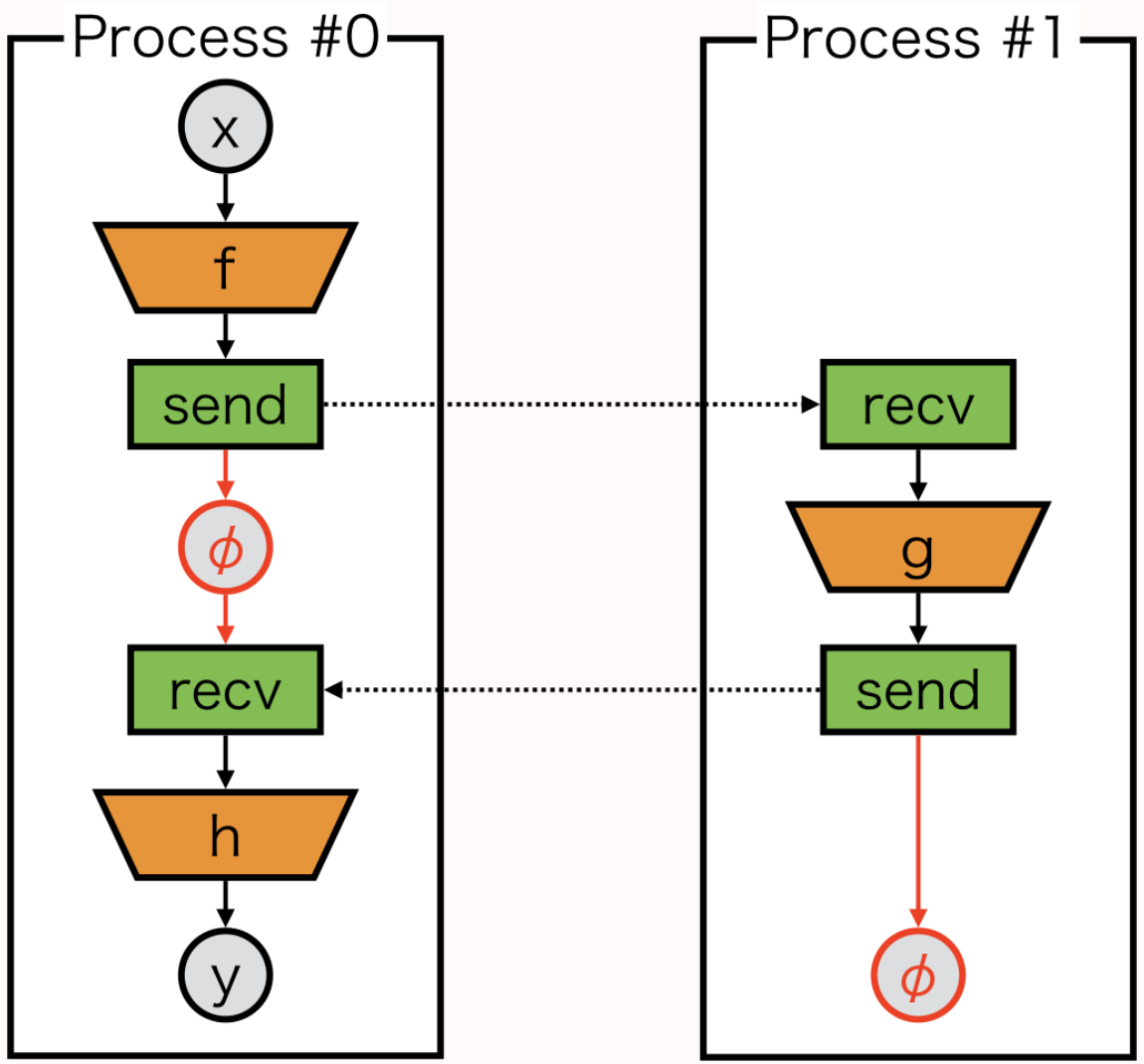
```
comm = chainermn.create_communicator()
device = comm.intra_rank  # use this device
optimizer = chainermn.create_multi_node_optimizer(…, comm)
```

Scaled to **V100x512** environment
(https://arxiv.org/abs/1809.00778)

Model parallelism

- Each node computes different part of the network (*model itself is parallelized*)
- MPI communication primitives *with backprop*

```
# rank 0
phi = send(x, comm, rank=1)
h = recv(comm, rank=1,
         delegate_variable=phi)

# rank 1
x = recv(comm, rank=0)
h = f(x)
phi = send(h, comm, rank=0)
```

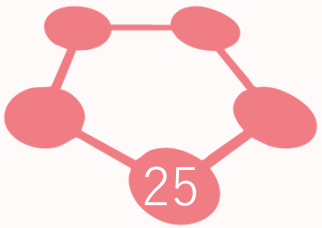

Process #0
x
f
send
ϕ
recv
h
y
Process #1
recv
g
send
ϕ

Model parallelism

- send returns a pseudo variable ϕ. It simulates the topology of full computational graph
- Collective communication routines, e.g. bcast, scatter, allgather etc., are also available

```
# rank 0
phi = send(x, comm, rank=1)
h = recv(comm, rank=1,
         delegate_variable=phi)
loss(h).backward()

# rank 1
x = recv(comm, rank=0)
h = f(x)
phi = send(h, comm, rank=0)
phi.backward()
```

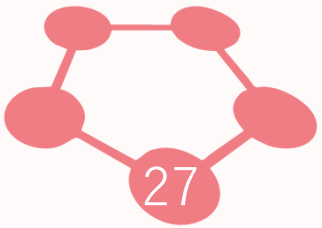
Domain specific add-on packages



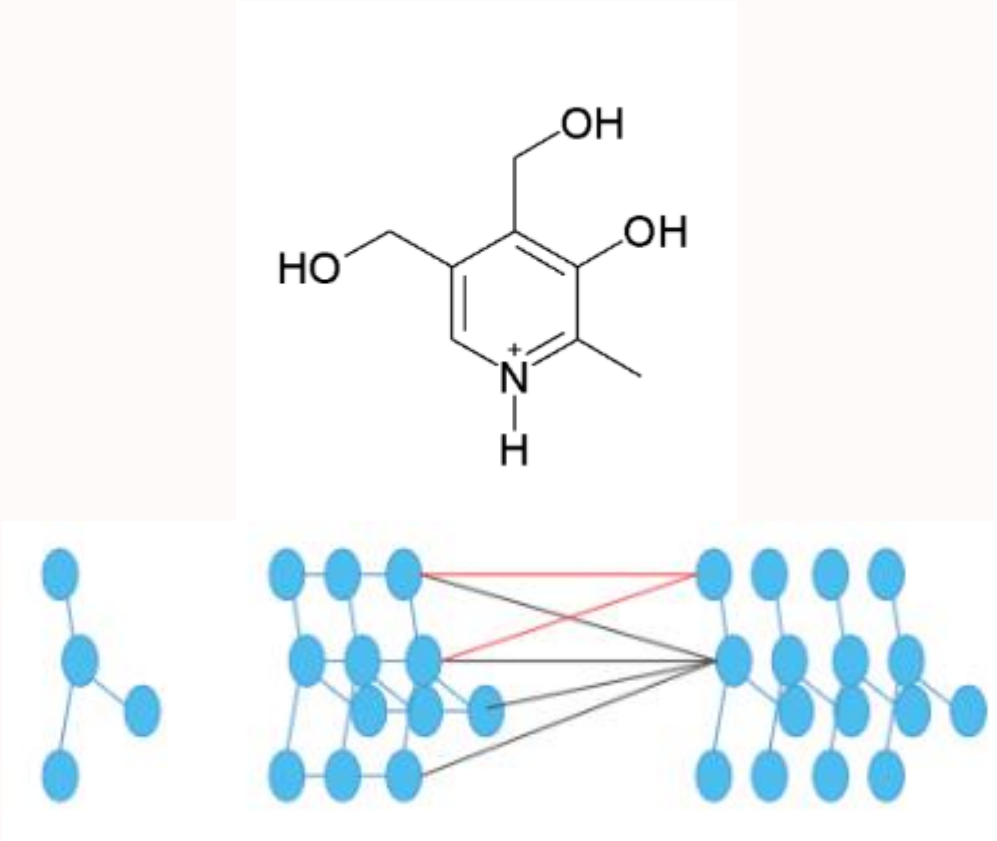
- ✓ Support standard computer vision tasks
  classification, object detection, semantic/instance segmentation
- ✓ Simple, unified interface
  easy to use and compose, optimized for computer vision workloads
- ✓ Guaranteed reproduction
  every method implemented is confirmed to reproduce the same performance as the original paper

Chainer RL

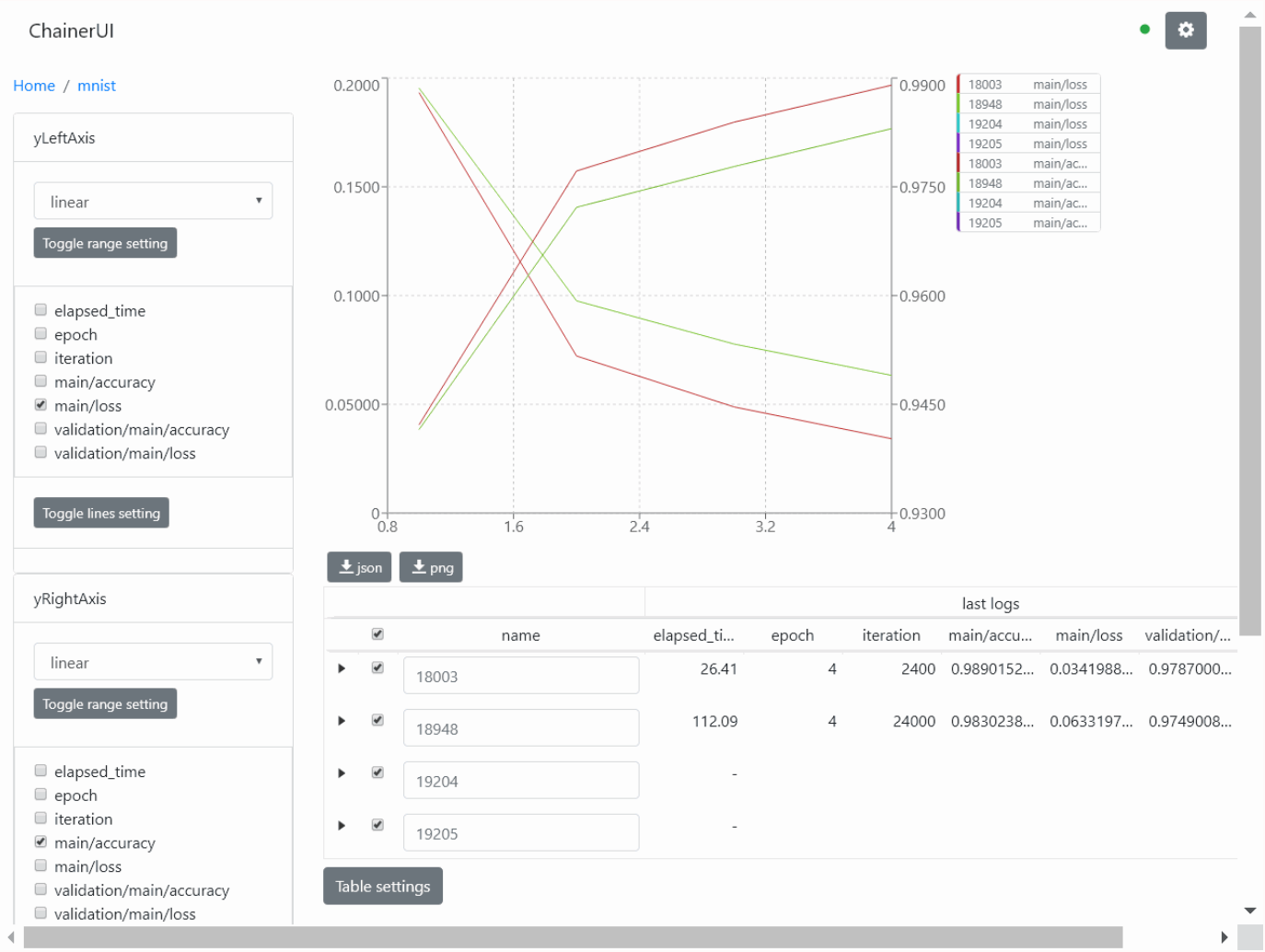
- ✓ Wide range of Deep RL methods covered
  DQN, Categorical DQN, IQN, DDPG, A3C, ACER, NSQ, PCL, PPO, TRPO
- ✓ Clean API and abstraction
  easy to combine multiple orthogonal design choices, e.g.
  discrete/continuous actions, recurrent models, async training, …
- ✓ Environment support
  compatible with OpenAI Gym interface



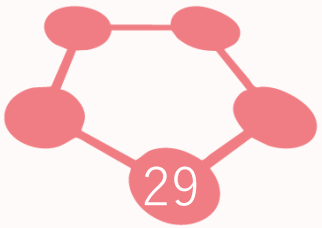
Chainer **Chemistry**



Chainer**UI**



What is needed for modern deep learning frameworks?



Speed

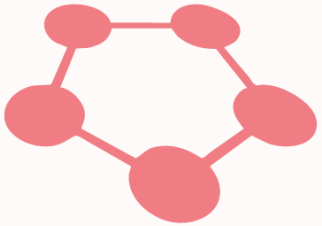
Faster trial-and-error
Larger scale

Environment support

Quick adoption of new hardware/environment

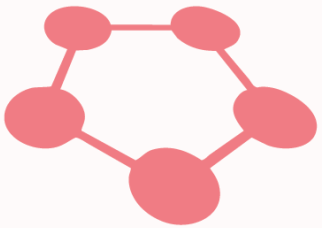
Quick Deployment

Quick application of research outcome



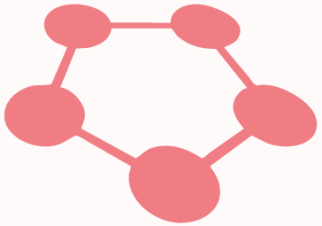
ChainerX

included in Chainer v6 beta1

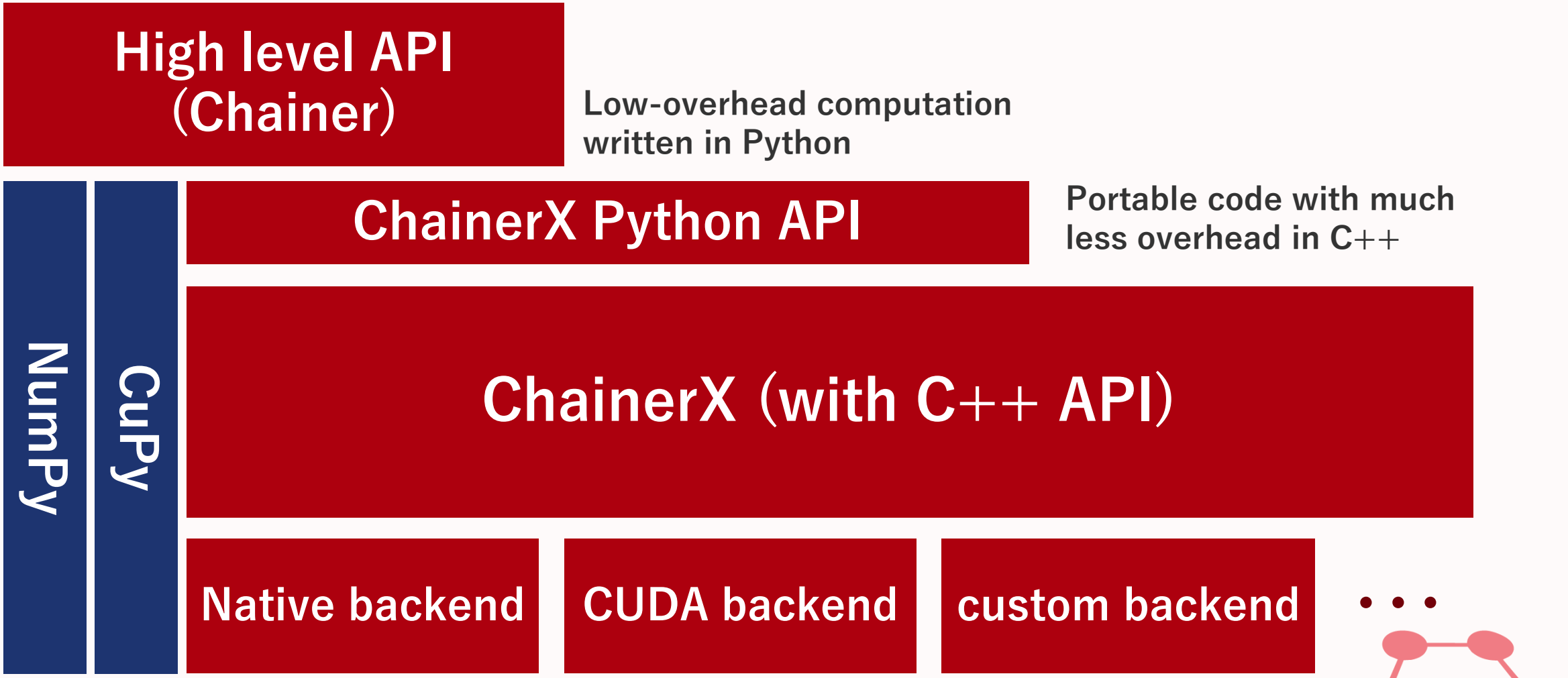


ChainerX = NumPy-like ndarray + autograd

- in C++ w/ a thin binding layer **Speed**
  = far less host-side overhead
- with pluggable device backends **Environment Support**
  = open to quickly add a new device support
- with pure C++ API **Quick Deployment**
  = available for Python-free native apps



Existing code using Chainer



ChainerX Python API: **chainerx** namespace

```
import chainerx as chx
x = chx.ones((2, 3),
             dtype=chx.float32,
             device='cuda:0')
y = (x + 1).require_grad()
z = chx.exp(y).sum()
z.backward()
```

- NumPy compatible API
- NN specific functions
  conv, batch_norm, …
- Device support
- require_grad() to make array differentiable

Chainer on ChainerX

```
arr = chx.ones((2, 3),
               dtype=chx.float32)
x = chainer.Variable(arr)

y = model(x)
y.backward()
```

- Wraps chx.ndarray with Variable
- FunctionNode fallbacks the computation to NumPy/CuPy
- Uses ChainerX (C++) computational graph with lower overhead in backprop

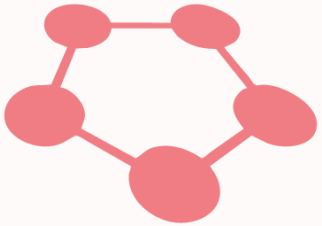
ChainerX C++ API

```
chainerx::Array x = chainerx::ones(
    {2, 3},
    chainerx::Dtype::kFloat32,
    chainerx::GetDevice("cuda:0"));
chainerx::Array y = (x + 1).RequireGrad();
chainerx::Array z = chainerx::Exp(y).Sum();
chainerx::Backward(z);
```

- Has almost one-to-one mapping to Python API
- Runs without CPython environment

Framework/API	Time per iteration (=fwd+bwd+update, msec)
Chainer on NumPy	14.48
Chainer on ChainerX	**7.54**
ChainerX Python	**1.88**
PyTorch	2.45

Host logic overhead



ChainerX: Roadmap

v6 — **May. 2019**

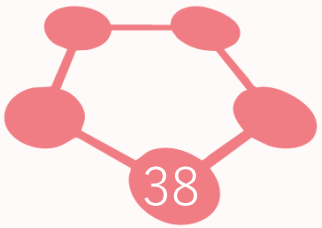
ChainerX
Basic ops
Integration to Chainer

v7 — **Nov. 2019**

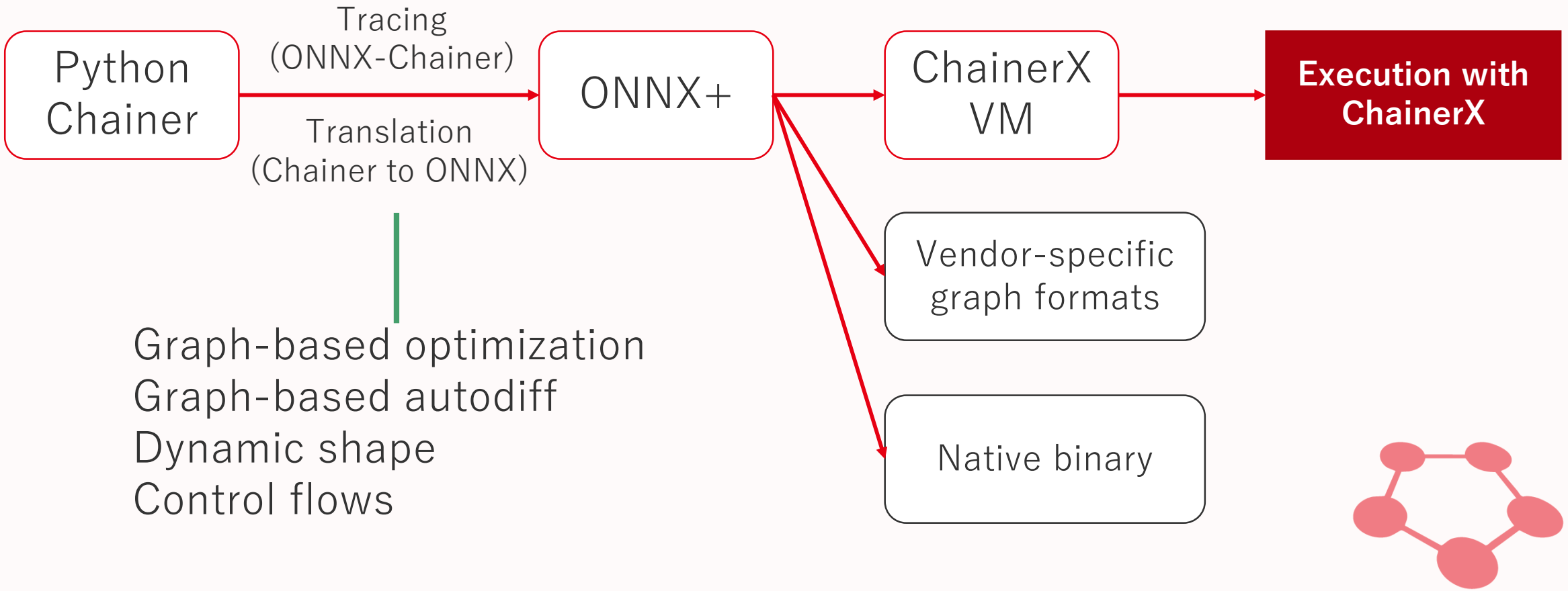
Wide coverage of ops
Ready for most users
C++ API made more accessible

future — **2020+**

Easier deploy
Wider coverage of “compiled models”



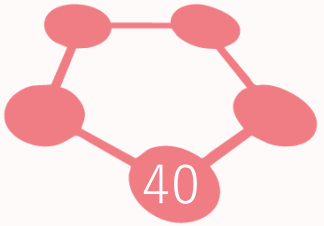
Chainer Compiler
https://github.com/pfnet-research/chainer-compiler

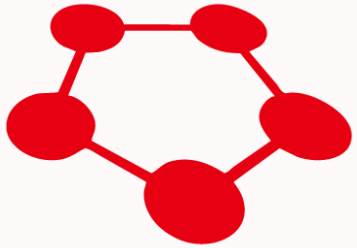


NUMFOCUS

OPEN CODE = BETTER SCIENCE

Google Summer of Code





Chainer

> Pioneering define-by-run API design

> Being made faster and more portable with ChainerX and Chainer Compiler

WE ARE HIRING!

@ChainerOfficial on Twitter

https://bit.ly/join-chainer-slack

https://preferred-networks.jp/en/jobs