Operationalizing PyTorch Models Using ONNX and ONNX Runtime

Spandan Tiwari (Microsoft)

Emma Ning (Microsoft)

Agenda

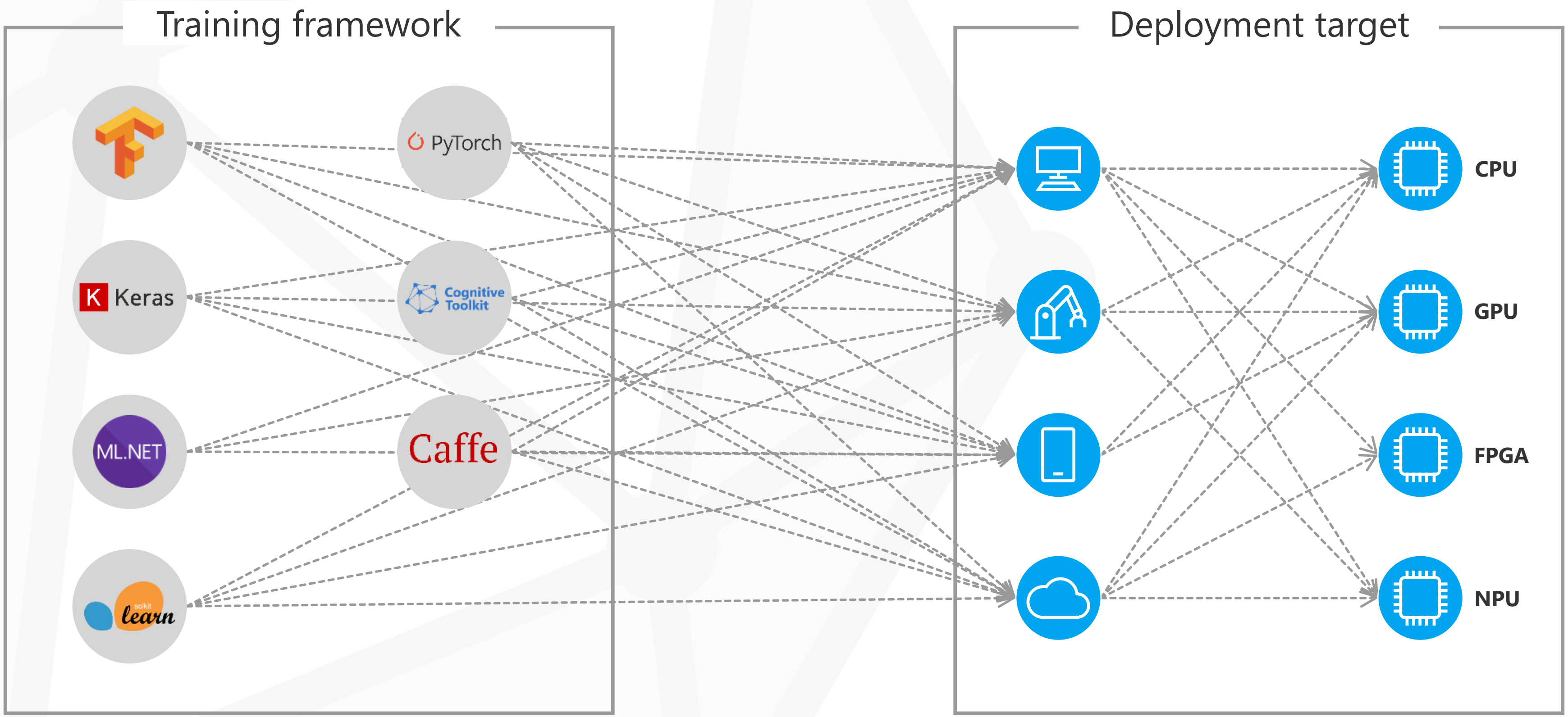
ONNX overview

Model operationalization with ONNX

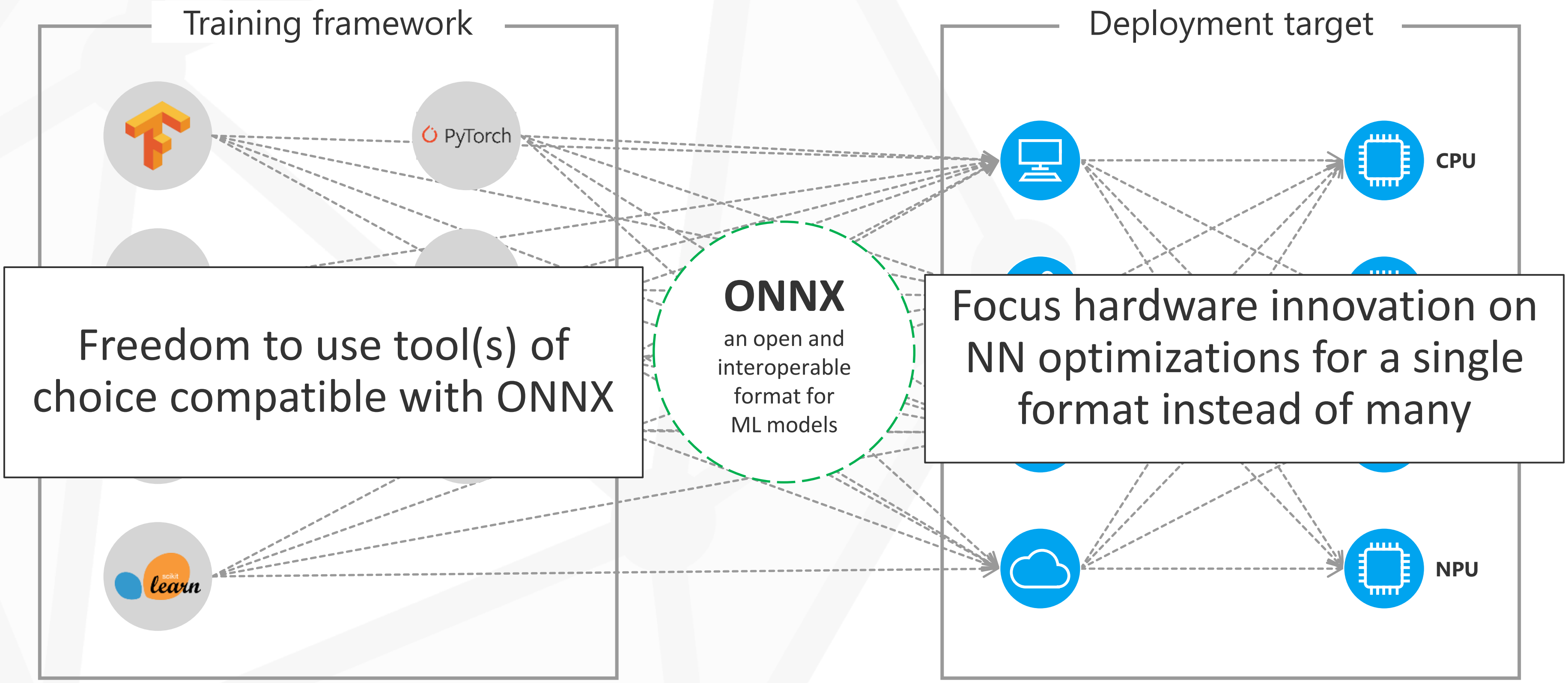
- Pytorch – ONNX exporter
- ONNX Runtime
- OLive

ONNX Overview

Problem - Training frameworks **x** Deployment targets



ONNX: an open and interoperable format for ML models



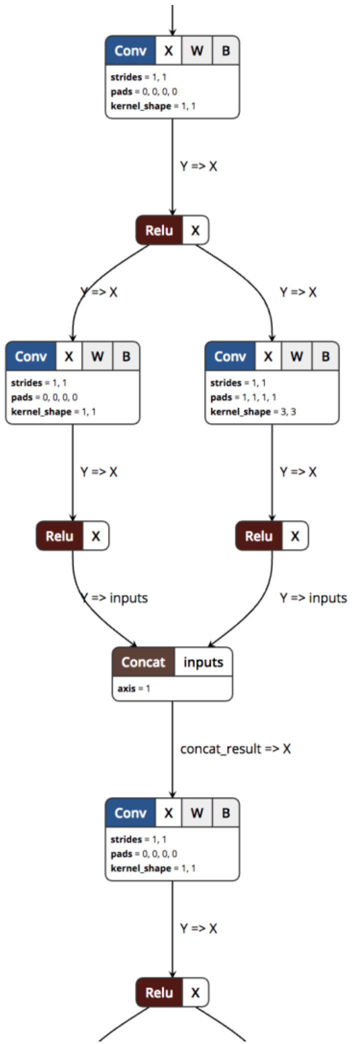
Freedom to use tool(s) of choice compatible with ONNX

Focus hardware innovation on NN optimizations for a single format instead of many

ONNX
- Open Neural Network Exchange

A specification that defines a standard format for ML models

- Consisting of:
    - common Intermediate Representation
    - full operator spec
- Model = graph composed of computational nodes
- Supports both DNN and traditional ML
- Backward compatible with comprehensive versioning



ONNX Community

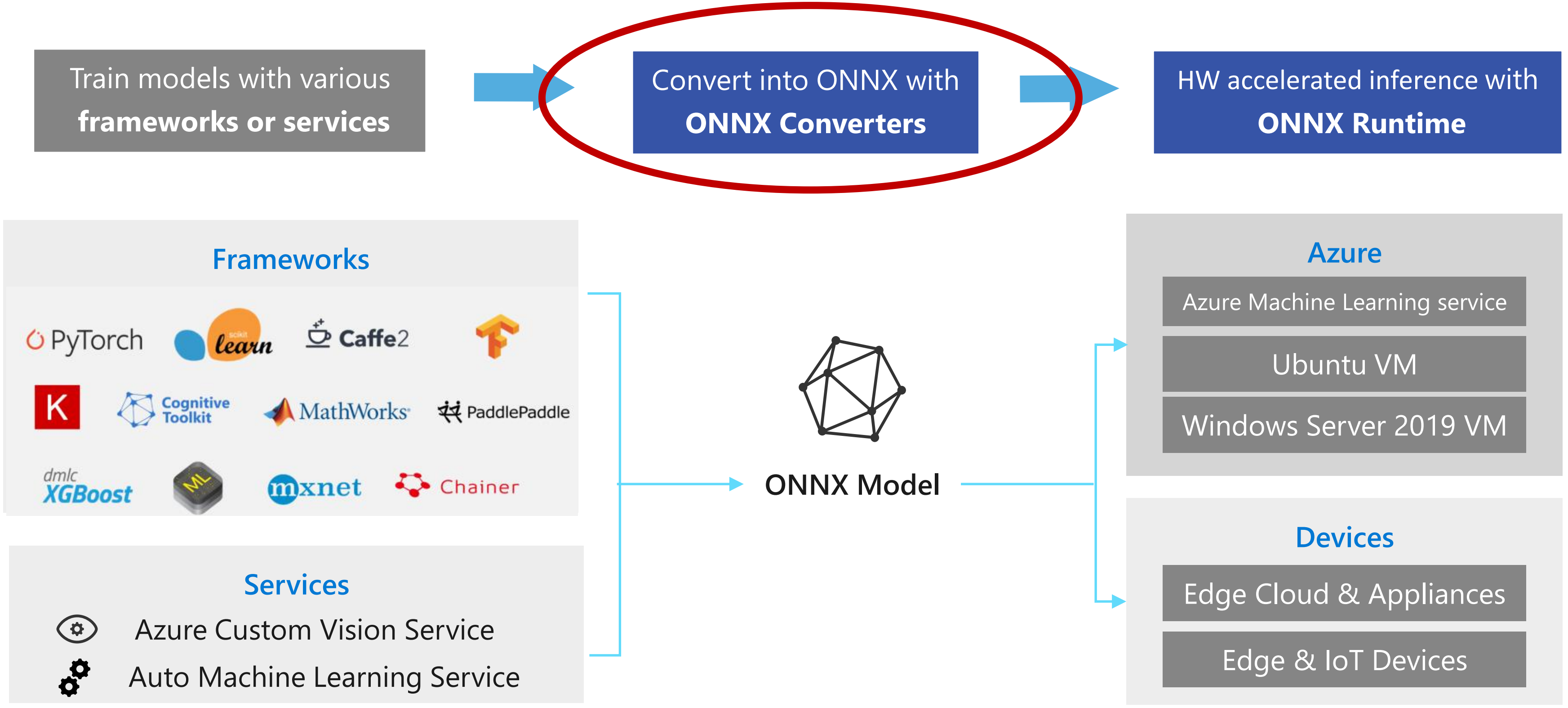
Alibaba Group 阿里巴巴集团
AMD
arm
aws
Bai du 百度
BITMAIN
CEVA
Facebook Open Source
GRAPHCORE
habana
Hewlett Packard Enterprise
HUAWEI
IBM
Idein Inc
intel AI
MathWorks
MAXAR
MEDIATEK
Microsoft
Neural Network Libraries
mi
NVIDIA
NXP
Oath: A Verizon company
OctoML
Preferred Networks
Qualcomm
sas
skymizer
SYNOPSYS
Tencent
unity

“We are pleased to welcome ONNX to the LF AI Foundation. We see ONNX as a key project in the continued growth of open source AI.”

- Mazin Gilbert, Chair of the LF AI Foundation Governing Board

Model operationalization with ONNX

Model operationalization with ONNX



Conversion - Open Source converters for popular frameworks

Tensorflow: onnx/tensorflow-onnx

PyTorch (native export)

Keras: onnx/keras-onnx

Scikit-learn: onnx/sklearn-onnx

CoreML: onnx/onnxmltools

LightGBM: onnx/onnxmltools

LibSVM: onnx/onnxmltools

XGBoost: onnx/onnxmltools

SparkML (alpha): onnx/onnxmltools

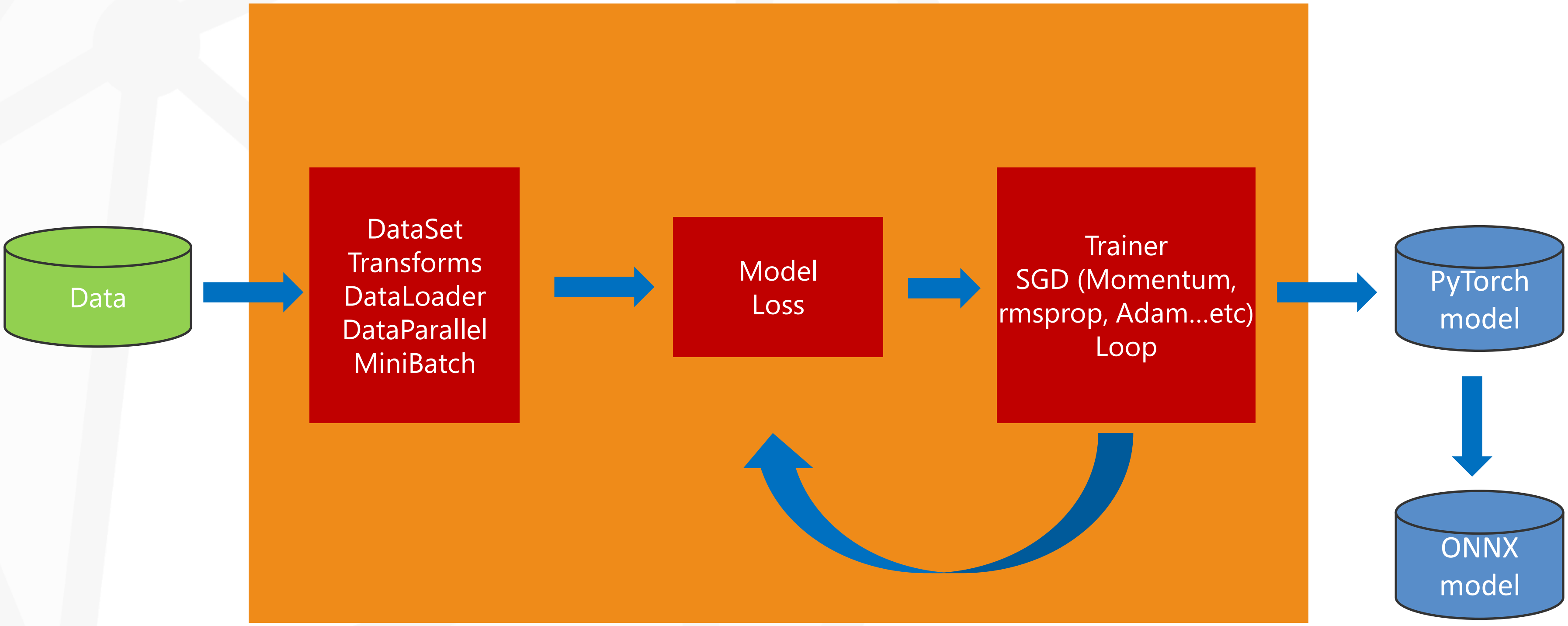
CNTK (native export)

PyTorch to ONNX Export

Overview

- PyTorch has native support for ONNX export
- Microsoft partners with Facebook on ONNX development in PyTorch
- PyTorch is easy to use and debug
- High performance without losing its flexibility
- Dynamic graph: ability to create complex topology that depends on the input data
- Community is large and growing...

PyTorch → ONNX Workflow



Writing a Model in PyTorch: Model Definition

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Set all your Module based layers in the `__init__`

Wire your model given input `x`

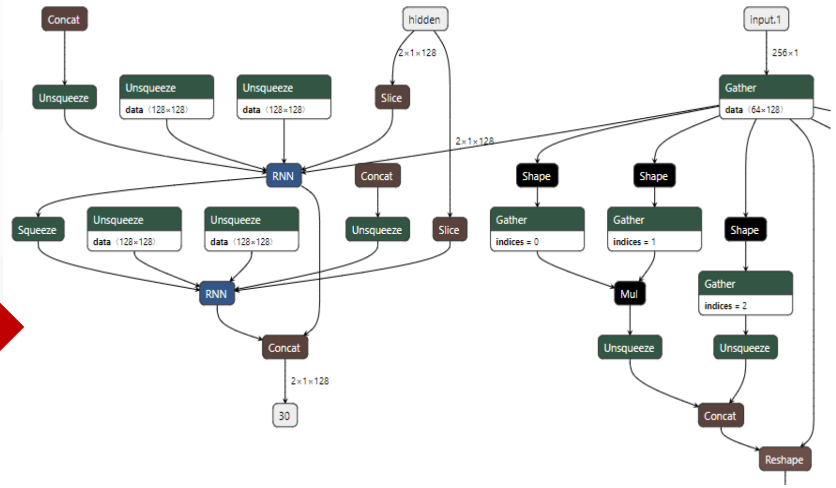
Writing a Model in PyTorch: Training Loop

```
model = Net().to(device)
# Use SGD with momentum
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# Set the model to train mode
model.train()

# Training loop
for epoch in range(epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

PyTorch to ONNX

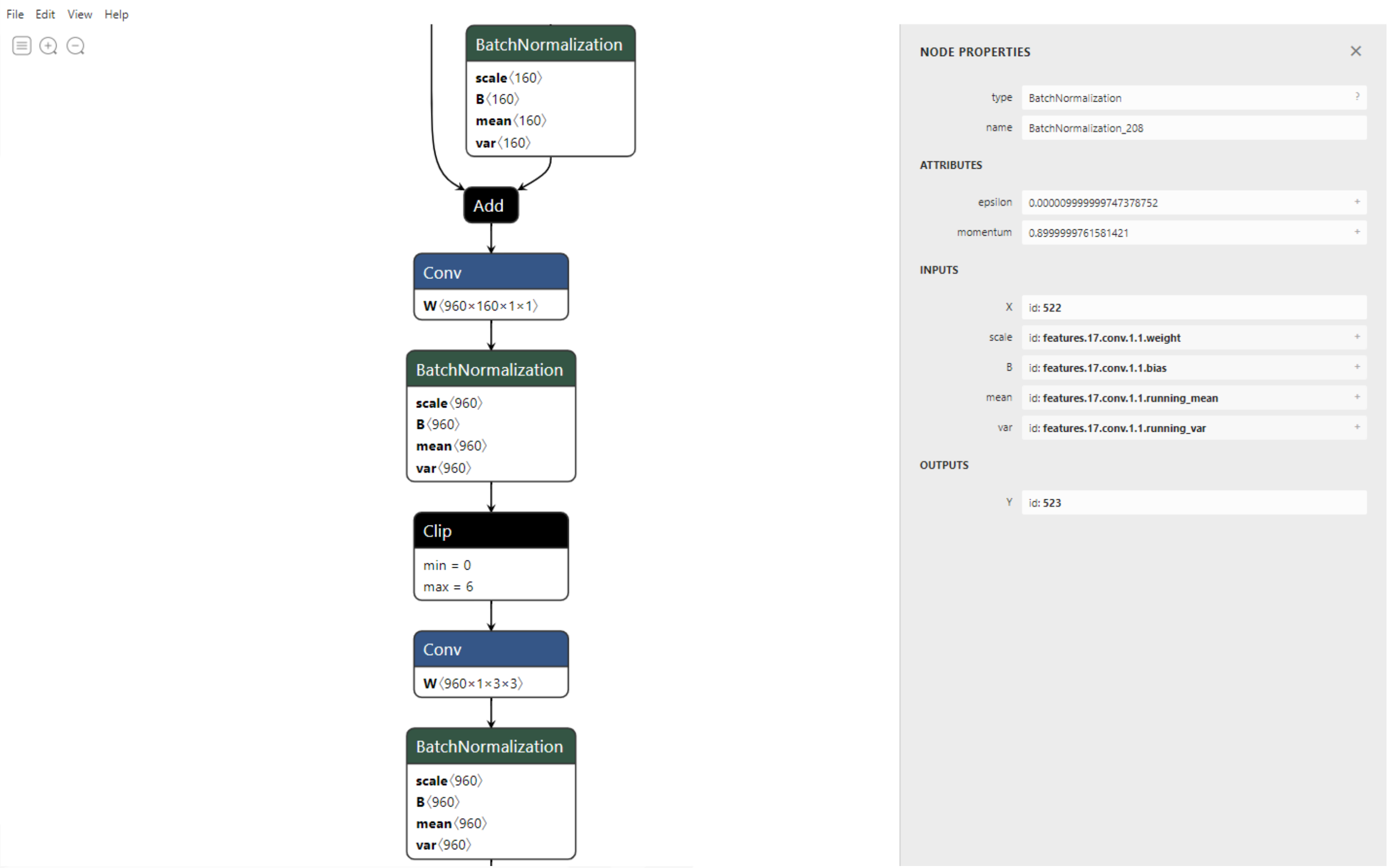
```python
from torch.autograd import Variable
import torch.onnx
import torch.nn as nn

class RNNModel(nn.Module):
    """Container module with an encoder, a recurrent module."""

    def __init__(self, numT, numInputs, numHidden, numLayers, dropout=0.5):
        super(RNNModel, self).__init__()
        self.drop = nn.Dropout(dropout)
        self.encoder = nn.Embedding(numT, numInputs)
        self.rnn = nn.RNN(numInputs, numHidden, num_layers=numLayers, dropout=dropout)

    def forward(self, input, hidden):
        embedding = self.drop(self.encoder(input))
        output, hidden = self.rnn(embedding, hidden)

model = RNNModel(numT=64, numInputs=128, numHidden=128, numLayers=2, dropout=0.5)
```

```python
input = torch.randn(1, 20, numInputs)

torch.onnx.export(model, input, "model.onnx")
```



ONNX Model Viewer: Netron

https://github.com/lutzroeder/netron



PyTorch ONNX Export API

torch.onnx.export(**model**,
input_args,
filename,
input_names=None, **output_names**=None,
opset_version=None,
do_constant_folding=True,
dynamic_axes=None,
keep_initializers_as_inputs=None,
enable_onnx_checker=True,
use_external_data_format=False)

PyTorch ONNX Export API

export(model, input_args, filename, …

```
from torch.autograd import Variable
import torch.onnx
import torch.nn as nn

class RNNModel(nn.Module):
    """Container module with an encoder, a recurrent module."""

    def __init__(self, numT, numInputs, numHidden, numLayers, dropout=0.5):
        super(RNNModel, self).__init__()
        self.drop = nn.Dropout(dropout)
        self.encoder = nn.Embedding(numT, numInputs)
        self.rnn = nn.RNN(numInputs, numHidden, num_layers=numLayers, dropout=dropout)

    def forward(self, input, hidden):
        embedding = self.drop(self.encoder(input))
        output, hidden = self.rnn(embedding, hidden)

model = RNNModel(numT=64, numInputs=128, numHidden=128, numLayers=2, dropout=0.5)
```

PyTorch ONNX Export API

export(model, input_args, filename, …

- Caller provides an example input to the model.
- Input could be a *torch.tensor*, for single input.
- For multiple inputs, provide a list or tuple.

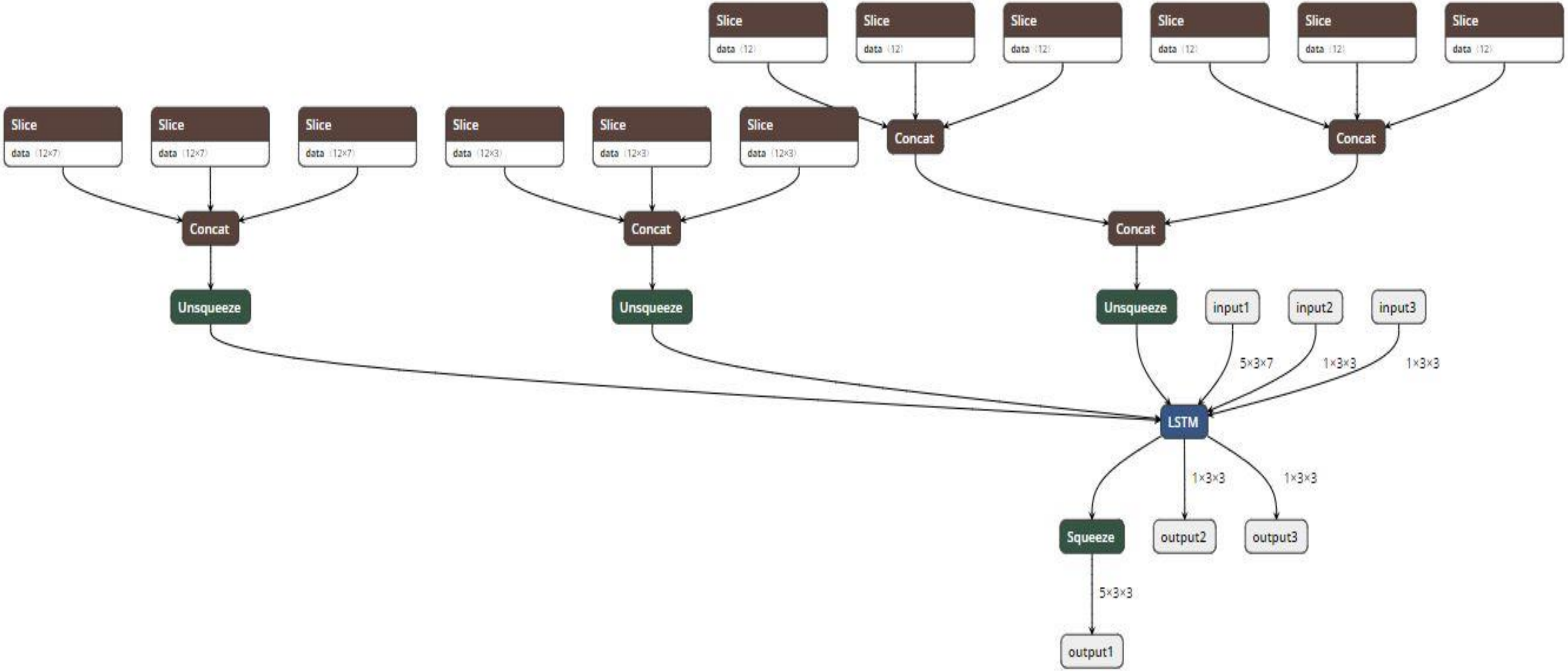
```
input = torch.randn(seq_len, batch_size, input_size)

h0 = torch.randn(num_layers*num_directions, batch_size, hidden_size)

c0 = torch.randn(num_layers*num_directions, batch_size, hidden_size)

torch_out = torch.onnx.export(model, (input, (h0, c0)), 'model.onnx')
```

PyTorch ONNX Export API

export(…, do_constant_folding=True, …

- PyTorch exporter can create graph with “extra” nodes.
- For example, weight format difference between PyTorch and ONNX RNNs.
- ONNX *W[iofc]* (input, output, forget, cell) vs. PyTorch uses *W[ifco]* (input, forget, cell, output)
- In some cases, variable batch-size accommodation

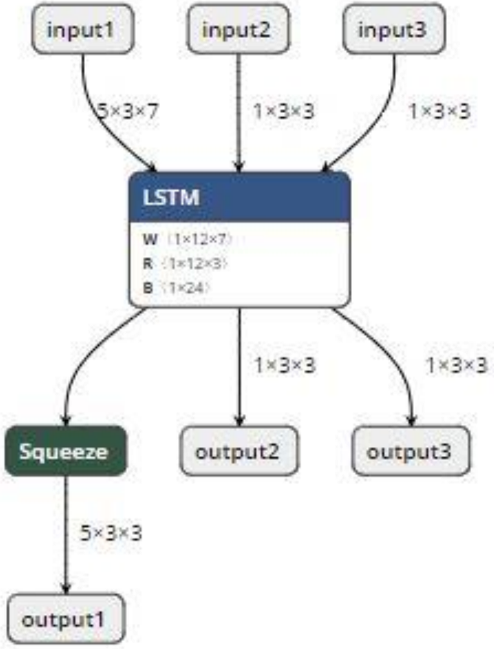
PyTorch ONNX Export API



PyTorch ONNX Export API

export(…, do_constant_folding=True, …

- Constant folding is a graph optimization.
- Does one-time computation on leaf ops with constant inputs and “folds” or replaces them with single constant.
- This reduces the graph size and reduces execution time.



PyTorch ONNX Export API

Model	Number of ops (Original model)	Number of ops (Constant-folded model)	Speedup (ORT CPU Execution Provider)
Bing AGI Encoder v4	147	98	~2.5x
Speech NNLM	104	53	~3.5x
PyTorch BERT (base)	1424	1184	10-12%

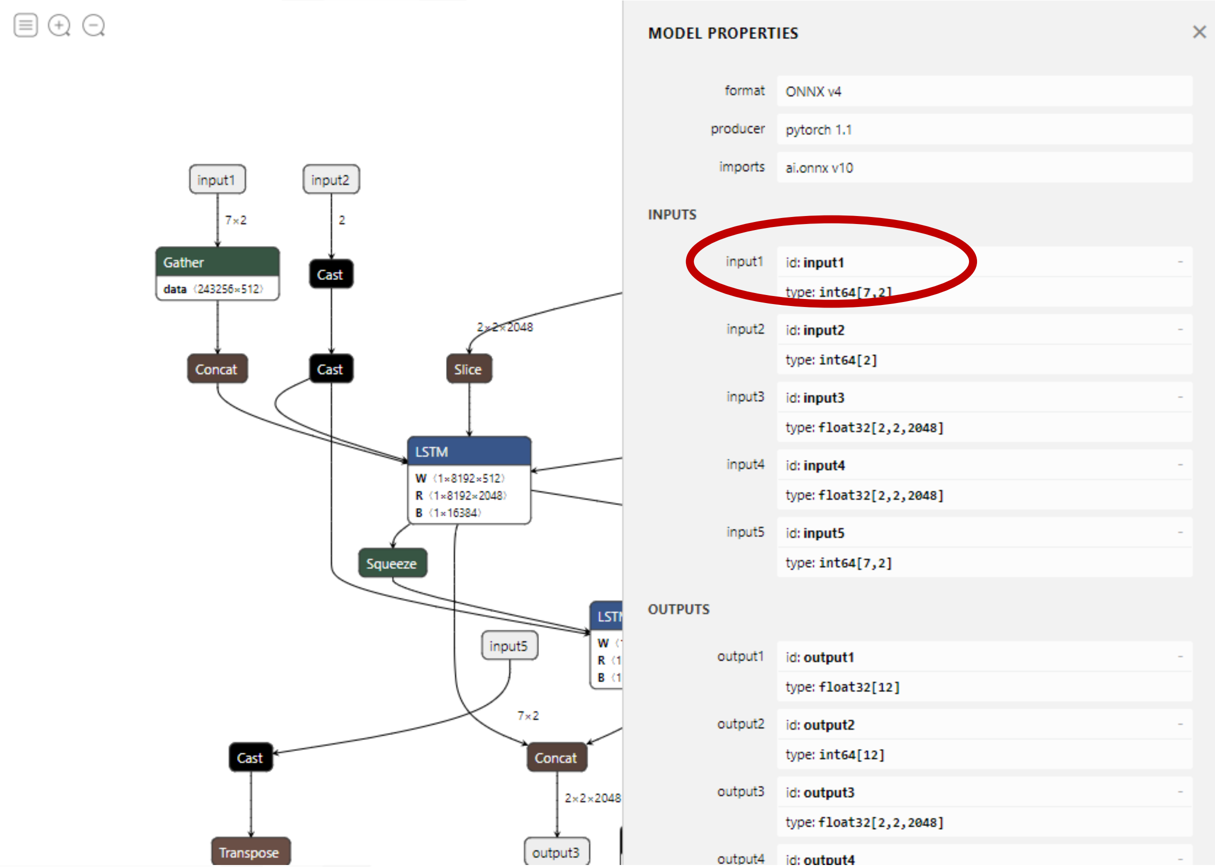
PyTorch ONNX Export – Variable-length Axes

export(…, dynamic_axes={}, …

- In many scenarios, the size of the input may be variable
  - Example: Batch axis for batch inference.
  - Example: Sequence axis case of RNN models
  - Example: Image size in FasterRCNN (object detection) models
- A variable-length axis can be represented in ONNX model
  - It is represented as a “string” dimension in ONNX
  - Each string represents a placeholder “value” for a length of the axis
  - Same string for different axes means that the length the axes must be the same for any input
- API supports specifying variable-length axes
  - Specified as arguments of top-level export API

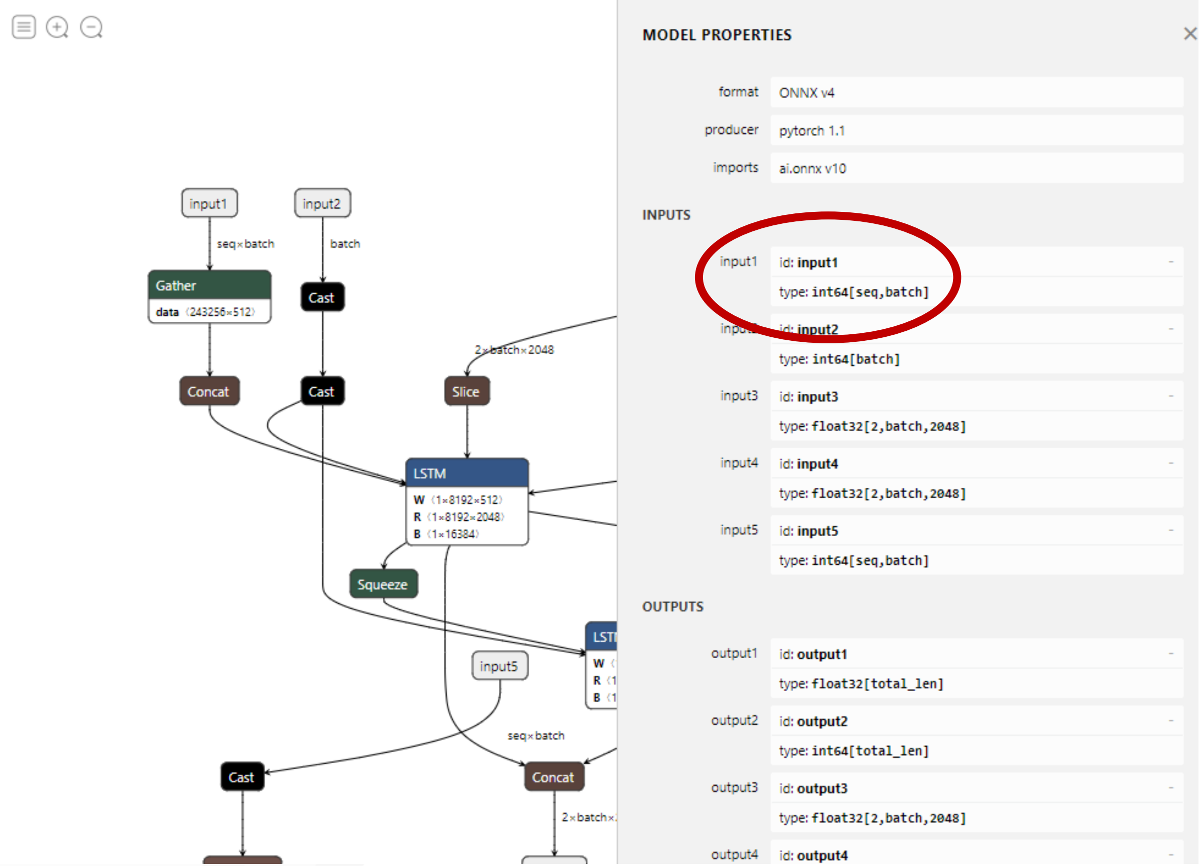
PyTorch ONNX Export – Variable-length Axes

ONNX model with fixed-length axes



PyTorch ONNX Export – Variable-length Axes

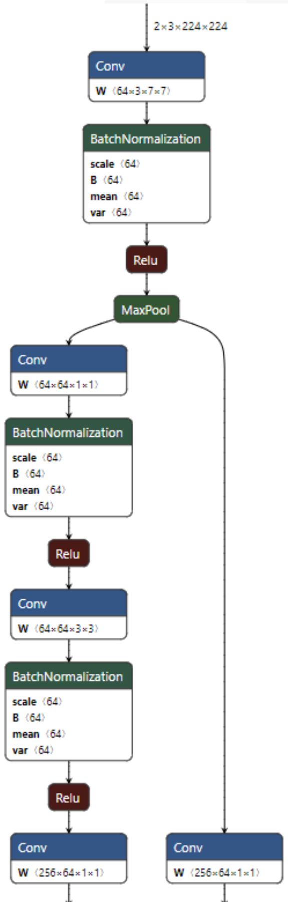
ONNX model with variable-length axes



PyTorch ONNX Export – Resnet50 Export

```
import torch
import torchvision

dummy_input = torch.randn(10, 3, 224, 224)
model = torchvision.models.resnet50(pretrained=True)

input_names = [ "input1" ]
output_names = [ "output1" ]

torch.onnx.export(model, dummy_input, "resnet50.onnx", verbose=True,
                  input_names=input_names, output_names=output_names,
                  do_constant_folding=True)
```

PyTorch ONNX Export – Resnet50 ONNX Model



PyTorch ONNX Export – Resnet50 ORT Inference

```python
import onnxruntime as rt
from PIL import Image

# Load and preprocess image
image = Image.open('TestElephant.jpg')
x = preprocessing(image)
x = x.numpy()

# Create ORT inference session and run inference
sess = rt.InferenceSession("resnet50.onnx")
result = sess.run([output_name], {input_name: x})
```

PyTorch ONNX Export – Resnet50 ORT Inference

PyTorch ONNX Export – Resnet50 ORT Inference

PyTorch ONNX – Deeper Look

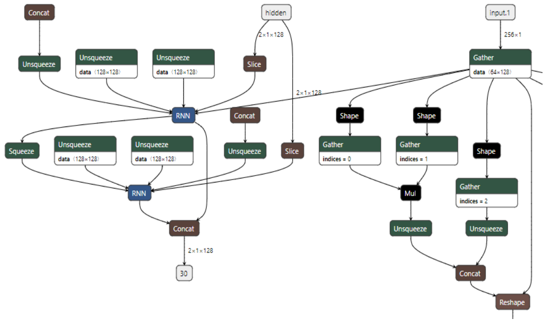
Underlying process for ONNX export

```python
from torch.autograd import Variable
import torch.onnx
import torch.nn as nn

class RNNModel(nn.Module):
    """Container module with an encoder, a recurrent module."""

    def __init__(self, numT, numInputs, numHidden, numLayers, dropout=0.5):
        super(RNNModel, self).__init__()
        self.drop = nn.Dropout(dropout)
        self.encoder = nn.Embedding(numT, numInputs)
        self.rnn = nn.RNN(numInputs, numHidden, num_layers=numLayers, dropout=dropout)

    def forward(self, input, hidden):
        embedding = self.drop(self.encoder(input))
        output, hidden = self.rnn(embedding, hidden)

model = RNNModel(numT=64, numInputs=128, numHidden=128, numLayers=2, dropout=0.5)
```

PyTorch JIT Compiler

Torch IR Graph

Torch IR graph to ONNX graph transformation



PyTorch ONNX – Code to Torch IR Graph

- Internally, there are two ways to convert PyTorch model to Torch IR graph
- This is implementation detail only – for ONNX export there's a single top-level API call, namely torch.onnx.export.

PyTorch ONNX – Tracing

- Structure of the model is captured by executing the model once using example inputs
- Records the flow of those inputs through the model

Pros

- No code change needed.
- More stable, well-supported

Cons

- Cannot support all models accurately, only those that use limited control-flow (conditionals or loops), no data-dependent control-flow.
- Does not capture control-flow, but just the sequence of on that single execution route.

PyTorch ONNX – Scripting

- Converting Python syntax directly to ScriptModule
- First Python AST is generated, the JIT compiler does semantic analysis and lowers it into a module

Pros

- Supports all models, with all control-flow routes
- It is the preferred way going forward

Cons

- Needs code change (inherit from torch.jit.ScriptModule + torch.jit.script decorator for methods).
- Only a subset of Python is supported.

PyTorch ONNX – Tracing

```python
class LoopAdd(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        h = x
        for i in range(x.size(0)):
            h = h + 1
        return h

input_1 = torch.ones(3, 16)
model = LoopAdd()
traced_model = torch.jit.trace(model, (input_1, ))

print(traced_model.graph)
```

PyTorch ONNX – Tracing

```
graph(%h.1 : Float(3, 16)):
  %4 : Long() = prim::Constant[value={1}](), scope: LoopAdd
  %5 : int = prim::Constant[value=1](), scope: LoopAdd
  %h.2 : Float(3, 16) = aten::add(%h.1, %4, %5), scope: LoopAdd
  %7 : Long() = prim::Constant[value={1}](), scope: LoopAdd
  %8 : int = prim::Constant[value=1](), scope: LoopAdd
  %h : Float(3, 16) = aten::add(%h.2, %7, %8), scope: LoopAdd
  %10 : Long() = prim::Constant[value={1}](), scope: LoopAdd
  %11 : int = prim::Constant[value=1](), scope: LoopAdd
  %12 : Float(3, 16) = aten::add(%h, %10, %11), scope: LoopAdd
  return (%12)
```

```
input_1 = torch.ones(5, 16)
print(np.all(np.array_equal(model(input_1),traced_model(input_1))))

>> False
```

PyTorch ONNX – Scripting

```
class LoopAdd(torch.jit.ScriptModule):
    def __init__(self):
        super().__init__()

    @torch.jit.script_method
    def forward(self, x):
        h = x
        for i in range(x.size(0)):
            h = h + 1
        return h

input_1 = torch.ones(3, 16)
model = LoopAdd()
traced_model = torch.jit.trace(model, (input_1, ))

print(traced_model.graph)
```

PyTorch ONNX – Scripting

```
graph(%0 : Float(3, 16)):
  %1 : bool = prim::Constant[value=1](), scope: LoopAdd
  %2 : int = prim::Constant[value=0](), scope: LoopAdd
  %3 : int = prim::Constant[value=1](), scope: LoopAdd
  %4 : int = aten::size(%0, %2), scope: LoopAdd
  %h : Float(*, *) = prim::Loop(%4, %1, %0), scope: LoopAdd
    block0(%i : int, %7 : Float(*, *)):
      %h.1 : Float(*, *) = aten::add(%7, %3, %3), scope: LoopAdd
      -> (%1, %h.1)
  return (%h)
```

```
input_1 = torch.ones(5, 16)
print(np.all(np.array_equal(model(input_1), traced_model(input_1))))

>> True
```

PyTorch ONNX – Final Thoughts

- Custom PyTorch operators can be exported to ONNX.
- Scenario: Custom op implemented in C++, which is not available in PyTorch.
- If equivalent set of ops are in ONNX, then directly exportable and executable in ORT.
- If some ops are missing in ONNX, then register a corresponding custom op in ORT.
- PyTorch has several ops, and some may not be exportable today.
- More details available at: https://pytorch.org/docs/stable/onnx.html

Model operationalization with ONNX

Train models with various **frameworks or services** → Convert into ONNX with **ONNX Converters** → HW accelerated inference with **ONNX Runtime**

Frameworks

PyTorch
scikit learn
Caffe2
K
Cognitive Toolkit
MathWorks
PaddlePaddle
dmlc XGBoost
ML
mxnet
Chainer

Services

Azure Custom Vision Service
Auto Machine Learning Service

ONNX Model

Azure

Azure Machine Learning service
Ubuntu VM
Windows Server 2019 VM

Devices

Edge Cloud & Appliances
Edge & IoT Devices

ONNX Runtime

A brief history

Problems:

- Teams using different frameworks, none with strong inference
- Teams building their own inference solutions
- Teams spending months to rewrite Python models into C++ code
- Optimizations developed by one team not accessible to others

Solution:

- Common inference engine containing all the optimizations from across Microsoft that works with multiple frameworks and runs everywhere inference needed

Inference – open source ONNX Runtime

ONNX RUNTIME

a **high-performance inference engine** for machine learning models in the ONNX format

Flexible

Supports full ONNX-ML spec (v1.2-1.6)

Supports CPU, GPU, VPU

C#, C, C++, Java and Python APIs

Cross Platform

Works on
-Mac, Windows, Linux
-x86, x64, ARM

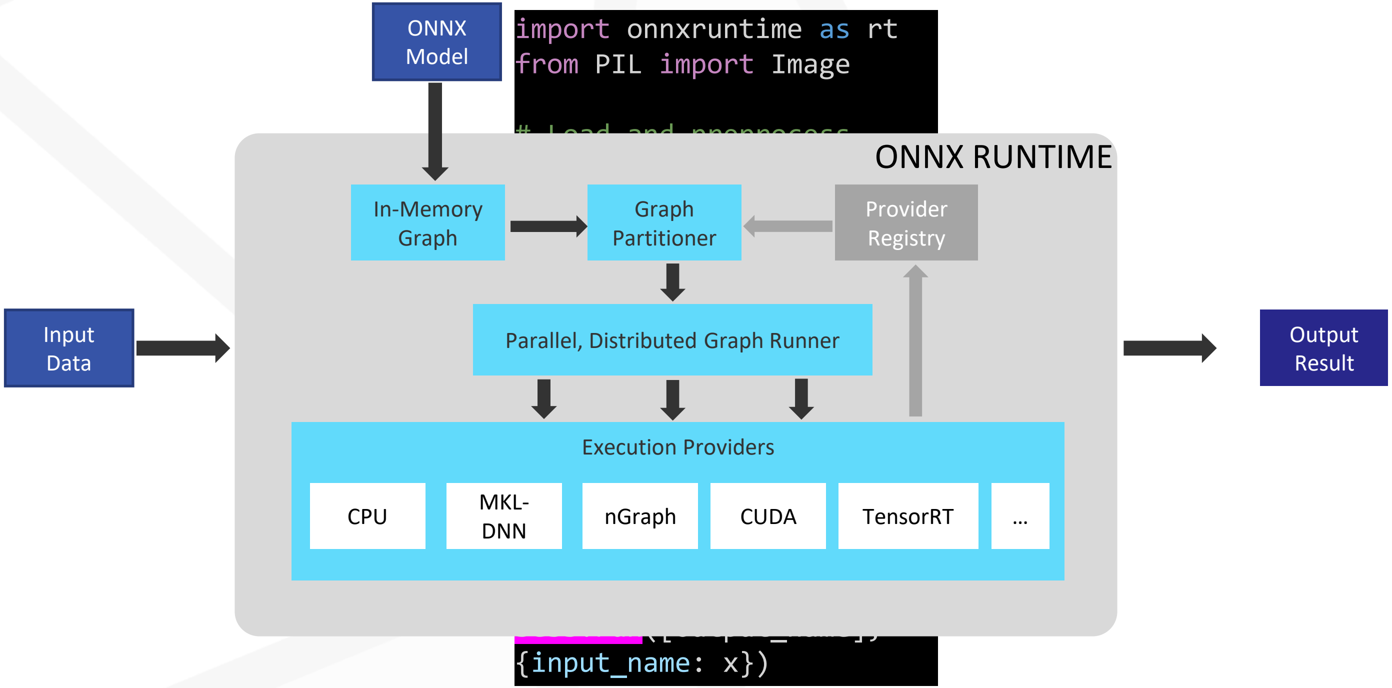
Also built-in to Windows 10 natively (WinML)

Extensible

Extensible architecture to plug-in optimizers and **hardware accelerators**

github.com/microsoft/onnxruntime

Leverages and abstracts hardware accelerators



BERT With ONNX Runtime (Bing/Office)

Apply BERT model to **every Bing search query globally** making Bing results more relevant and intelligent
-> latency and cost challenges

ORT Inferences Bing's 3-layer BERT with 128 sequence length

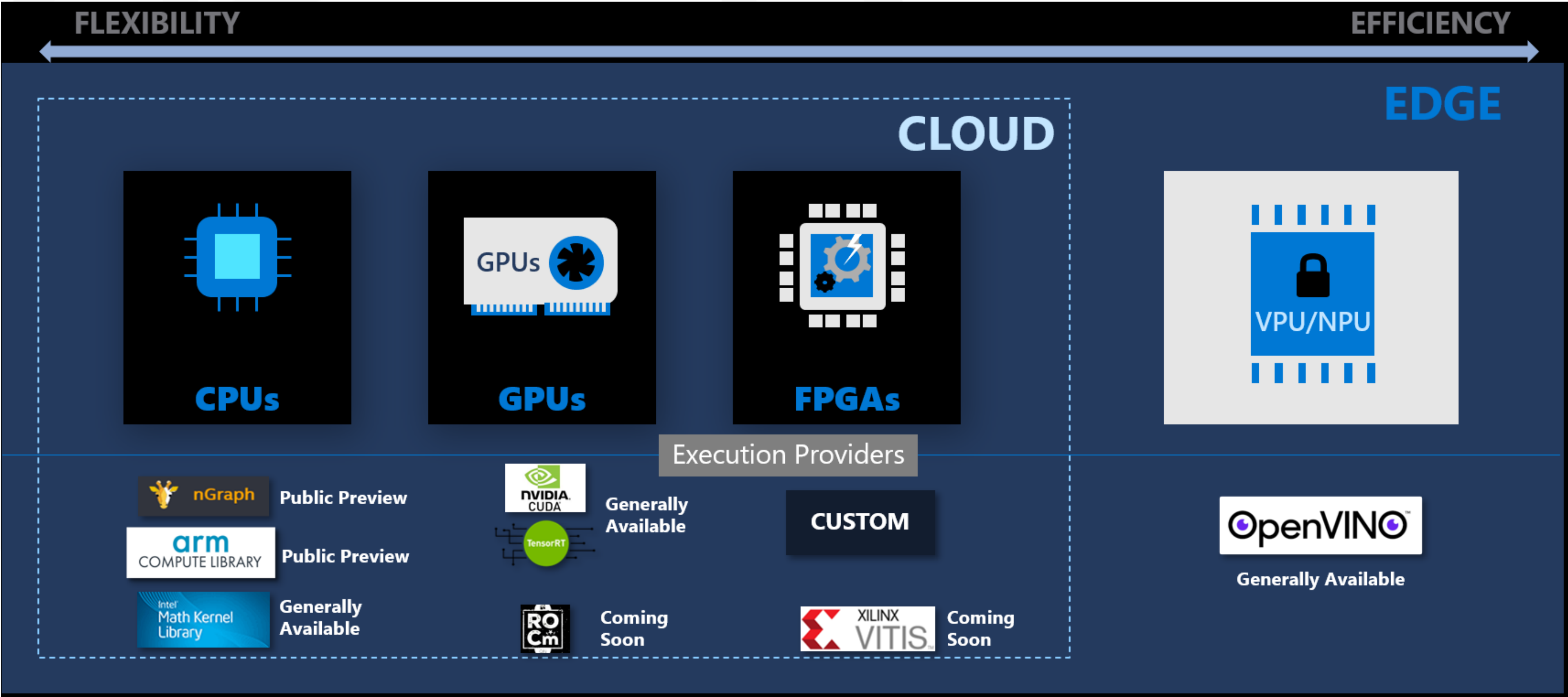
- On CPU, 17x latency speed up with ~100 queries per second throughput.
- On NVIDIA GPUs, more than 3x latency speed up with ~10,000 queries per second throughput on batch size of 64

ORT inferences BERT-SQUAD with 128 sequence length and batch size 1 on Azure Standard NC6S_v3 (GPU V100)

- in 1.7 ms for 12-layer fp16 BERT-SQUAD.
- in 4.0 ms for 24-layer fp16 BERT-SQUAD.

		Batch size	Inference on	Throughput (Query per second)	Latency (milliseconds)
CPU	Original 3-layer BERT	1	Azure Standard F16s_v2 (CPU)	6	157
	ONNX Model	1	Azure Standard F16s_v2 (CPU) **with ONNX Runtime**	111	9
GPU	Original 3-layer BERT	4	Azure NV6 GPU VM	200	20
	ONNX Model	4	Azure NV6 GPU VM **with ONNX Runtime**	500	8
	ONNX Model	64	Azure NC6S_v3 GPU VM **with ONNX Runtime + System Optimization** (Tensor Core with mixed precision, Same Accuracy)	10667	6

ONNX Runtime HW Ecosystem

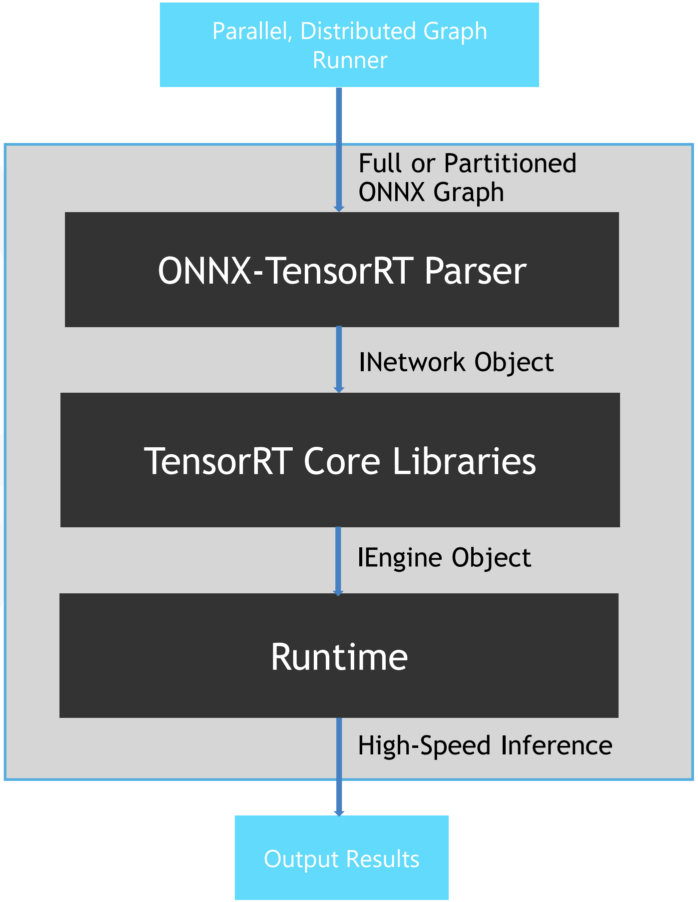


ONNX Runtime + TensorRT

TensorRT

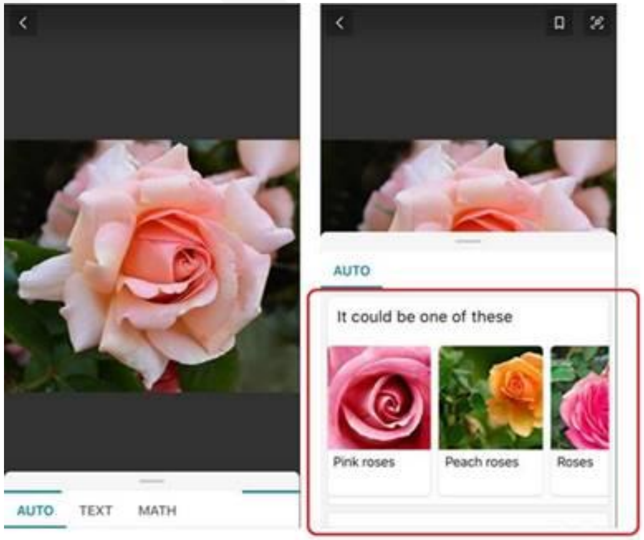
Platform for High-Performance Deep Learning Inference

- Maximize throughput for latency-critical apps with optimizer and runtime
- Optimize your network with layer and tensor fusions, dynamic tensor memory and kernel auto tuning
- Deploy responsive and memory efficient apps with INT8 & FP16 optimizations
- Fully integrated as a backend in ONNX runtime



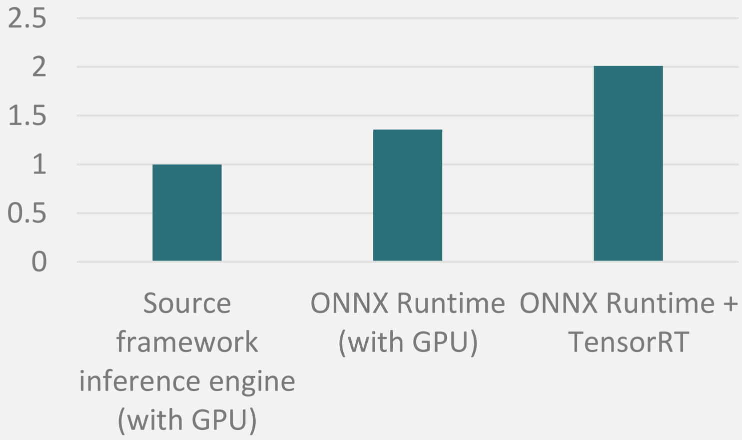
Multimedia with ONNX Runtime + TensorRT

Bing Visual Search- enables the ability to visually identify a flower from a picture, supplemented with rich information about the flower



PERFORMANCE

<u>**2x**</u> performance gain on ONNX Runtime with TensorRT



OLive

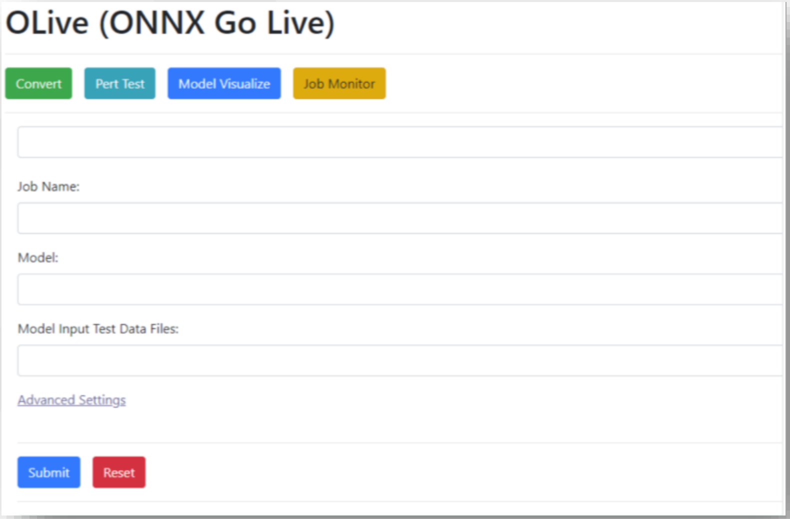
OLive

Simplify model operationalization with an easy-to-use pipeline for

- model conversion to ONNX
- performance optimization with ONNX Runtime

4 Ways to use OLive

- Use With Command Line Tool
- Use With Local Web App
- Use With Jupyter Notebook
- Use Pipeline With Kubeflow



https://github.com/microsoft/olive

Demo

Try it for yourself

- **ONNX** at

  https://github.com/onnx/onnx
- **Pytorch-ONNX exporter** at

  https://pytorch.org/docs/stable/onnx.html
- **ONNX Runtime** at

  https://github.com/microsoft/onnxruntime
- **TensorRT** Instructions at

  aka.ms/onnxruntime-tensorrt
- **OLive** at

  https://github.com/microsoft/olive