Deploying PyTorch Models for Real-time Inference On the Edge

Moritz August
CDO & Co-Founder
Nomitri

2021 embedded VISION summit
VIRTUAL | MAY 25-28

NOMITRI
PERCEPTION AI FOR MOBILE

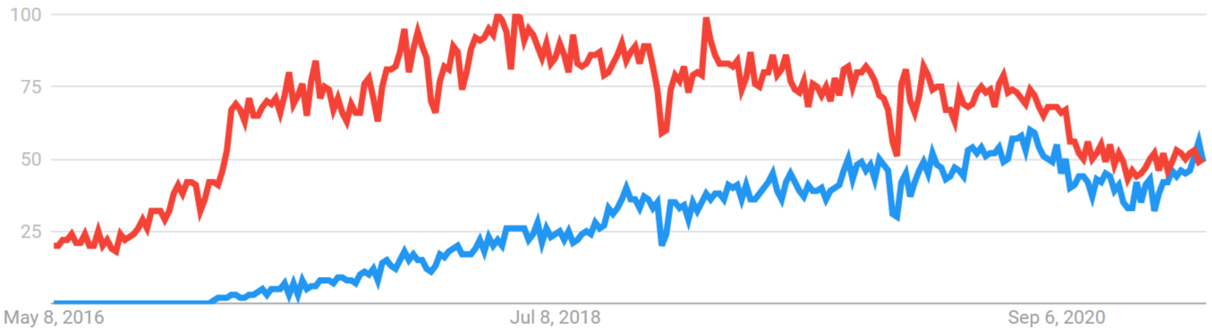
Intro

About us

- Start-Up based in Berlin, Germany
- Deep Learning vision applications for mobile/edge
- First product in retail
- From edge application over ML to backend service
- Use PyTorch and C++ library to deploy our models

Why PyTorch?

- Imperative, simple API
- Dynamic computation graphs at its core
- Great ecosystem
- Debug with Python debugger
- Caught up with TensorFlow



Google Trends: PyTorch (Blue) vs TensorFlow (Red)

PyTorch Mobile & Torchvision

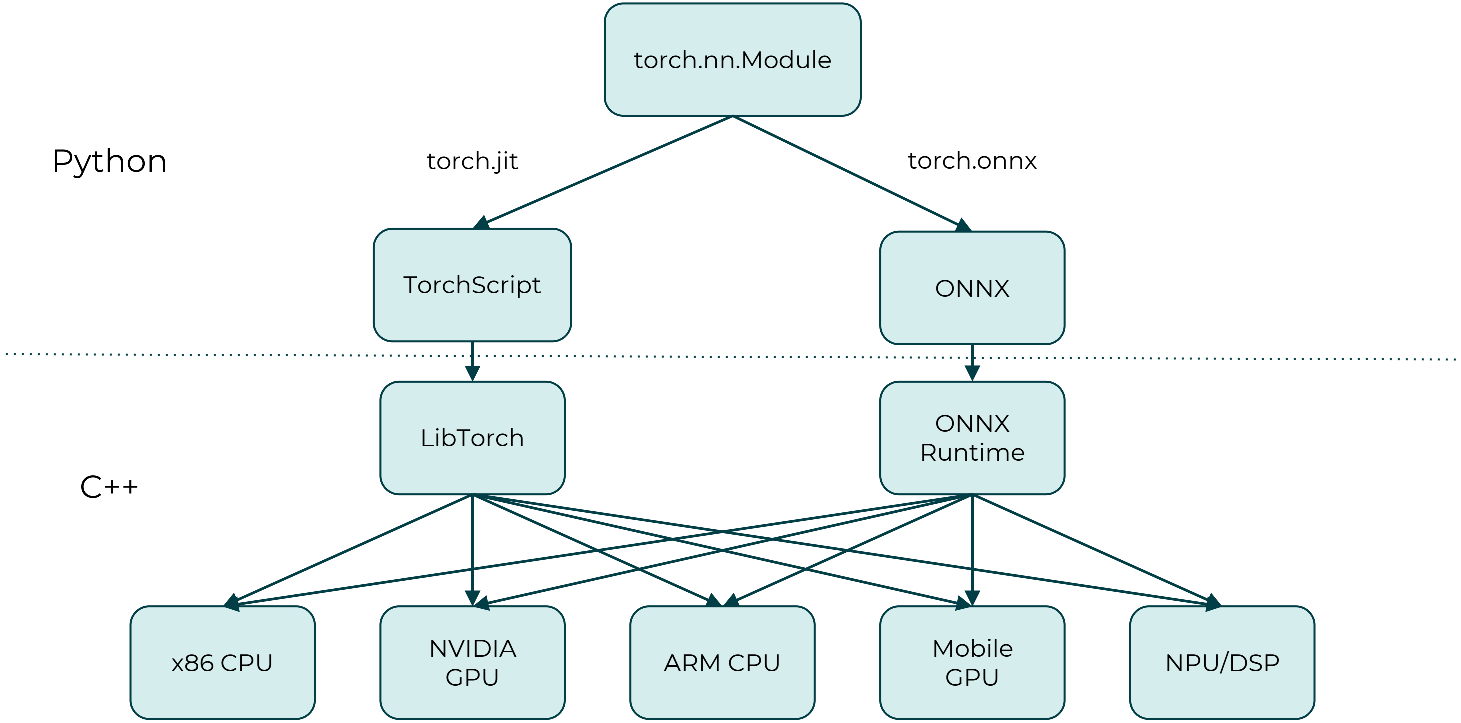
PyTorch Mobile

- Available for Android, Linux and iOS
- Provides several ready-to-use models
- Simple deployment workflows
- Support for Arm CPU and accelerators

Torchvision

- Provides many pre-trained vision architectures
- Tools for augmentation, IO and bounding boxes
- Contains models optimized for mobile/edge

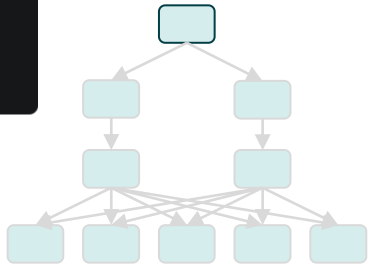
Deployment Workflows



Model Optimization

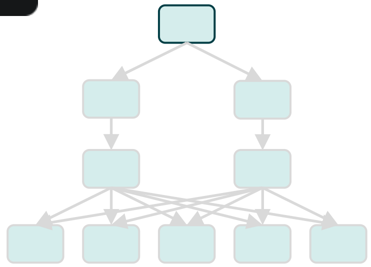
Example Model: Classification

```
class ToyClassifier(nn.Module):

    def __init__(self, block_params: List, n_features: int,
                 pool_kernel: Tuple[int, int]):
        super(ToyClassifier, self).__init__()

        blocks = [ConvBNReLU(in_channels=in_channels,
                             out_channels=out_channels,
                             stride=stride)
                  for (in_channels, out_channels, stride) in block_params]

        self.blocks = nn.ModuleList(blocks)
        self.pooling = nn.MaxPool2d(kernel_size=pool_kernel)
        self.classifier = nn.Linear(in_features=n_features, out_features=1000)
```



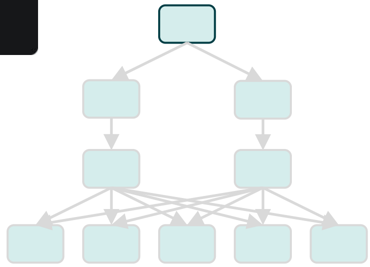
Example Model: Classification

```python
def forward(self, x: torch.Tensor) -> torch.Tensor:
    for block in self.blocks:
        x = block(x)
    features = self.pooling(x).reshape((1, -1))
    return self.classifier(features)
```



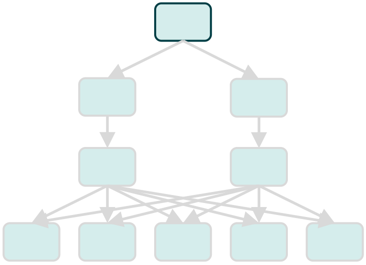
Example Model: Classification

```python
class ConvBNReLU(nn.Module):

    def __init__(self, in_channels: int, out_channels: int, stride: int = 1):
        super(ConvBNReLU, self).__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=(3, 3),
                      stride=(stride, stride), padding=(1, 1), bias=False),
            nn.BatchNorm2d(num_features=out_channels),
            nn.ReLU()
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.layers(x)
```



Architecture Optimization

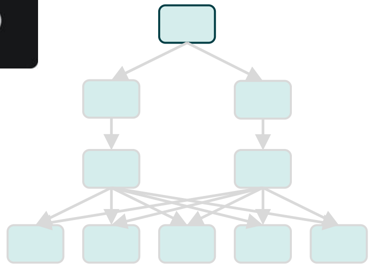
- Use depthwise-separable convolutions
- Fuse operations like Conv, BatchNorm, ReLU
- Make channels divisible by 8
- Use efficient and fuseable activations like ReLU
- Use channels-last memory format



Architecture Optimization

```
nn.Sequential(
    nn.Conv2d(in_channels=in_channels, out_channels=in_channels, kernel_size=(3, 3),
              stride=(stride, stride), padding=(1, 1), bias=False, groups=in_channels),
    nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=(1, 1),
              bias=False),
    nn.BatchNorm2d(num_features=out_channels),
    nn.ReLU()
)
```

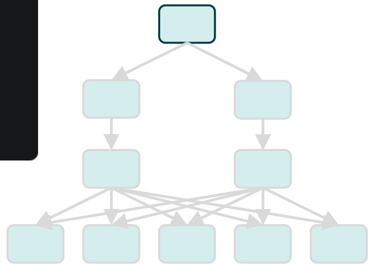
```
def fuse(self):
    torch.quantization.fuse_modules(self, ['layers.1', 'layers.2', 'layers.3'], inplace=True)
```



Architecture Optimization

Torchvision offers utility function to adapt channel numbers

```python
# ...

blocks = [OptimizedConvBNReLU(in_channels=3,
                              out_channels=_make_divisible(block_params[0][1], 8),
                              stride=block_params[0][2])]

for in_channels, out_channels, stride in block_params[1:]:
    blocks.append(OptimizedConvBNReLU(in_channels=_make_divisible(in_channels, 8),
                                      out_channels=_make_divisible(out_channels, 8),
                                      stride=stride)
                  )

self.blocks = nn.ModuleList(blocks)
self.pooling = nn.MaxPool2d(kernel_size=pool_kernel)
self.classifier = nn.Linear(in_features=_make_divisible(n_features, 8),
                            out_features=1000)
```



Model compression

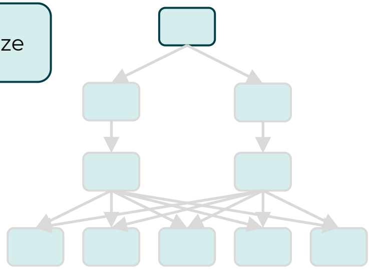
- PyTorch has built-in support for model compression
  - Pruning
  - Quantization
- Pruning is implemented via weight masks
- Quantization for weights and activations
- Advanced techniques have open-source implementations

Quantization

- Quantization supports different modes
  - Dynamic quantization
  - Static quantization
  - Quantization-aware training
- Has two backends for execution on x86 and Arm CPUs
- Quantization can be customized via configuration

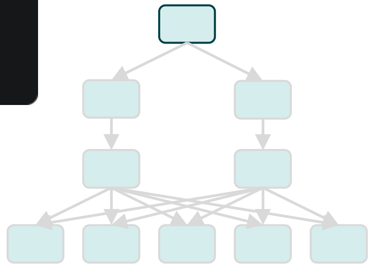
Workflow static/post-training quantization

Add Stubs → Train → Fuse → Prepare → Calibrate → Quantize



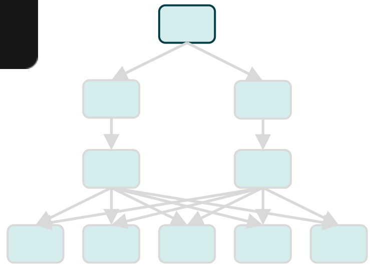
Quantization

```
class ToyClassifier(nn.Module):

    def __init__(self, block_params: List, n_features: int,
                 pool_kernel: Tuple[int, int]):
        super(ToyClassifier, self).__init__()

        self.quant = torch.quantization.QuantStub()
        self.dequant = torch.quantization.DeQuantStub()

        # ...
```



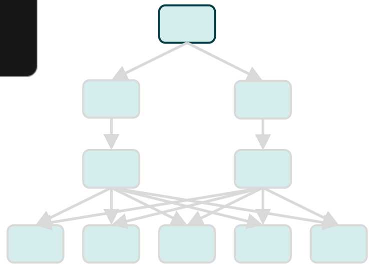
Quantization

```python
def forward(self, x: torch.Tensor) -> torch.Tensor:
    x = self.quant(x)
    for block in self.blocks:
        x = block(x)
    features = self.pooling(x).reshape((1, -1))
    logits = self.classifier(features)
    return self.dequant(logits)
```



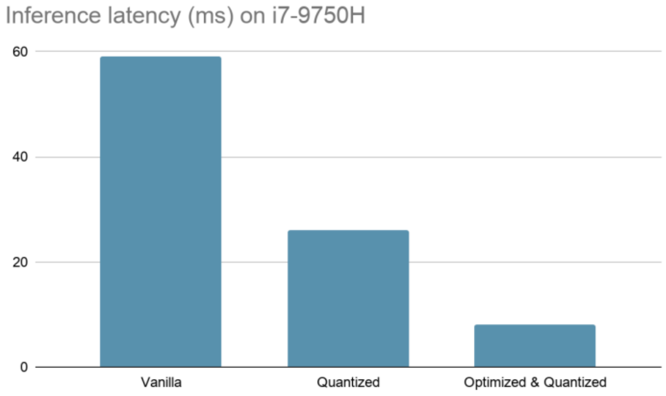
Quantization

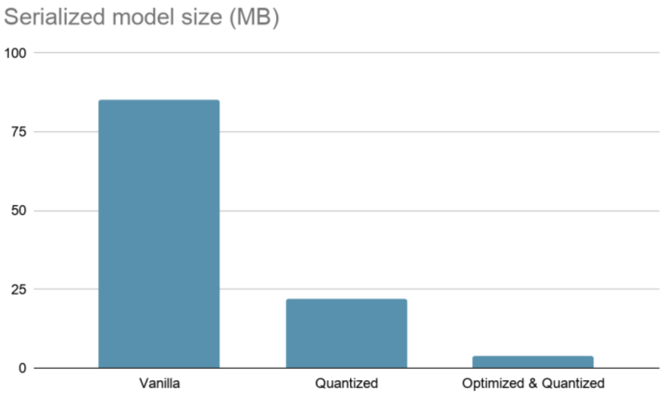
```python
def quantize_model(model: nn.Module, dataloader: DataLoader,
                   backend: str = 'qnnpack'):
    torch.backends.quantized.engine = backend
    model.qconfig = torch.quantization.get_default_qconfig(backend)

    model_prepared = torch.quantization.prepare(model)
    for sample in dataloader:
        model_prepared(sample)
    model_quantized = torch.quantization.convert(model_prepared)
    return model_quantized
```



Model Comparison

- Performing the optimizations yields drastic improvements
- Measurements for 8 layers with 18 to 1154 channels

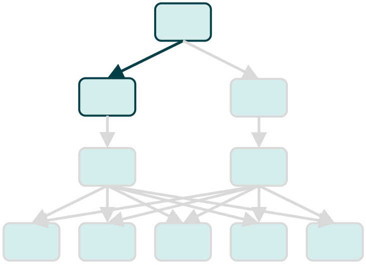




Deployment

TorchScript

- Statically typed intermediate representation
- Serialize and ship to production environments
- Multiple ways to convert models to TorchScript
  - Convert entire module
  - Trace graph with example input
  - Write in TorchScript



TorchScript: Conversion, Tracing & Serialization


```python
def script_and_serialize(model: nn.Module, path: str):
    scripted_model = torch.jit.script(model)
    torch.jit.save(scripted_model, path)
```

```python
def trace_and_serialize(model: nn.Module, path: str):
    example_input = torch.rand((1, 3, 224, 224))
    with torch.no_grad():
        traced_model = torch.jit.trace(model, example_input)
    torch.jit.save(traced_model, path)
```

TorchScript: Example output

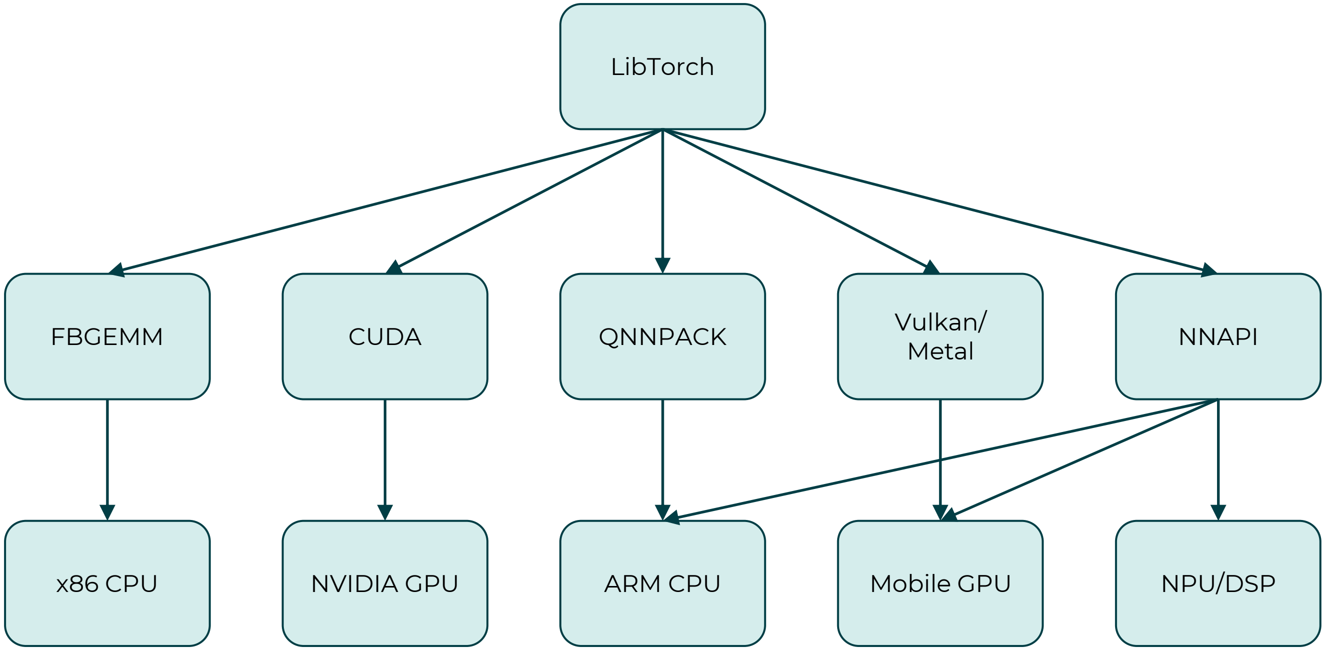
```
graph(%self : __torch__.torch.nn.modules.container.Sequential,
      %input.1 : Tensor):
  %2 : __torch__.torch.nn.modules.conv.Conv2d = prim::GetAttr[name="0"](%self)
  %3 : __torch__.torch.nn.modules.conv.___torch_mangle_0.Conv2d = prim::GetAttr[name="1"](%self)
  %4 : __torch__.torch.nn.modules.batchnorm.BatchNorm2d = prim::GetAttr[name="2"](%self)
  %5 : __torch__.torch.nn.modules.activation.ReLU = prim::GetAttr[name="3"](%self)
  %10 : int = prim::Constant[value=16]() # /usr/local/lib/python3.8/dist-packages/torch/nn/modules/conv.py:396:53
  %11 : int = prim::Constant[value=1]() # /usr/local/lib/python3.8/dist-packages/torch/nn/modules/conv.py:395:45
  %12 : Tensor = prim::GetAttr[name="weight"](%2)
  %13 : Tensor? = prim::GetAttr[name="bias"](%2)
  %14 : int[] = prim::ListConstruct(%11, %11)
  %15 : int[] = prim::ListConstruct(%11, %11)
  %16 : int[] = prim::ListConstruct(%11, %11)
  %input.3 : Tensor = aten::conv2d(%input.1, %12, %13, %14, %15, %16, %10) # /usr/local/lib/python3.8/dist-
packages/torch/nn/modules/conv.py:395:15
  # ...
  %input.7 : Tensor = aten::batch_norm(%input.5, %45, %46, %43, %44, %42, %33, %34, %26) #
/usr/local/lib/python3.8/dist-packages/torch/nn/functional.py:2149:11
  %input.9 : Tensor = aten::relu(%input.7) # /usr/local/lib/python3.8/dist-packages/torch/nn/functional.py:1206:17
  return (%input.9)
```

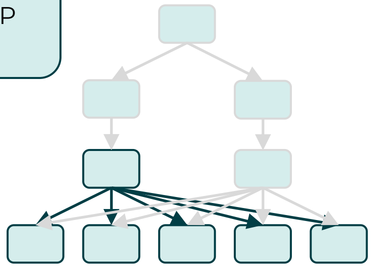
LibTorch

- PyTorch has C++ 14 API
- Closely follows the Python API
- Can be used by simple inclusion of torch header
- Typically used for inference only
- Simply load TorchScript models for inference

General workflow for using LibTorch

Backends

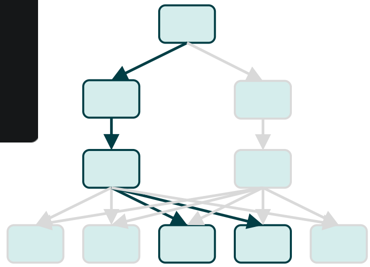

LibTorch
FBGEMM
CUDA
QNNPACK
Vulkan/
Metal
NNAPI
x86 CPU
NVIDIA GPU
ARM CPU
Mobile GPU
NPU/DSP



Mobile CPU and GPU

- Simple interface to perform graph optimizations
- Allows optimization for different backends
  - XNNPACK for floating point on Arm CPU
  - QNNPACK for quantized 8-bit on Arm CPU
  - Vulkan for GPU on Android
  - Metal for GPU on iOS

```python
def script_and_optimize(model: nn.Module, path: str, backend: str = 'CPU'):
    scripted_model = torch.jit.script(model)
    scripted_model = optimize_for_mobile(script_module=scripted_model,
                                         backend=backend) # 'CPU', 'Vulkan' or 'Metal'
    torch.jit.save(scripted_model, path)
```

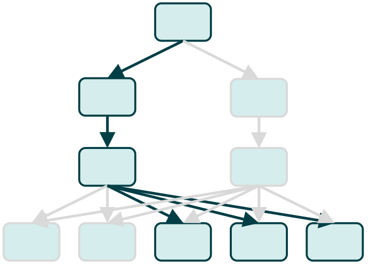


NNAPI

- TorchScript model can be optimized to run via NNAPI
- Model should be fused and quantized beforehand
- Channels-last memory format is mandatory
- NNAPI model can be wrapped to provide float interface

Workflow NNAPI optimization

Quantize → Remove Stubs → Prepare NNAPI input → Trace → Convert



NOMITRI
PERCEPTION AI FOR MOBILE

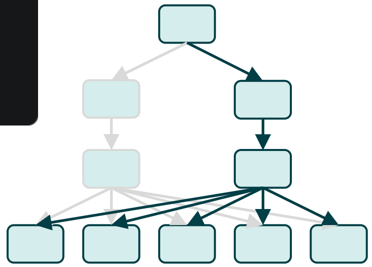
NNAPI

```python
def deploy_nnapi(input_float: Tensor, model_quantized: nn.Module,
                 quantizer: nn.Module, dequantizer: nn.Module):
    input_tensor = quantizer(input_float)

    input_tensor = input_tensor.contiguous(memory_format=torch.channels_last)
    input_tensor.nnapi_nhwc = True

    with torch.no_grad():
        model_quantized_traced = torch.jit.trace(model_quantized, input_tensor)
    nnapi_model = convert_model_to_nnapi(model_quantized_traced, input_tensor)
    nnapi_model_float_interface = torch.jit.script(
        torch.nn.Sequential(quantizer, nnapi_model, dequantizer))
```

ONNX

- Open Neural Network Exchange (ONNX)
- Open-Source ecosystem for switching frameworks
- ONNX Runtime ML acceleration framework by Microsoft
- Supports variety of accelerators for inference and training
- PyTorch has native support for export to ONNX

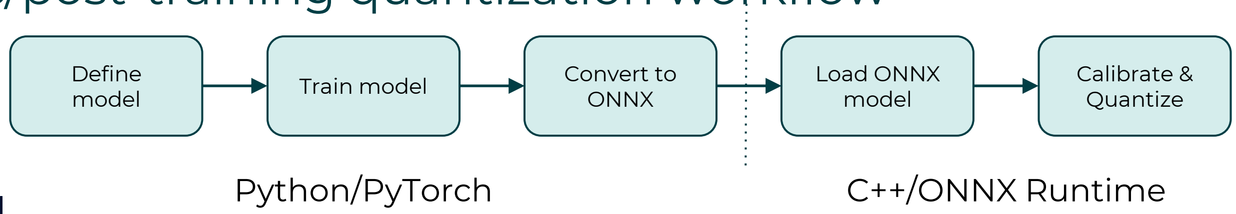
```
def export_onnx(model: nn.Module, path: str):
    example_input = torch.rand(1, 3, 224, 224)
    onnx.export(model=model, args=(example_input,), f=path,
                input_names=['input_batch'], output_names=['logits'],
                opset_version=12)
```

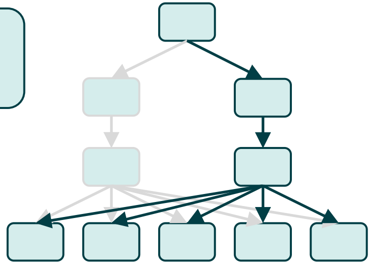


ONNX Quantization

- Only unquantized models can be exported to ONNX
- ONNX Runtime supports quantization
  - Dynamic
  - Static
  - Quantization-aware training
- PyTorch dataloader can be re-used with a small wrapper

Static/post-training quantization workflow



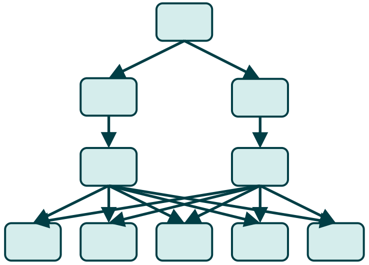


ONNX Quantization


```
def quantize_onnx(float_path: str, quantized_path: str,
                  dataloader: DataLoader):
    onnx_q_loader = ONNXQuantizationDataReader(quant_loader=dataloader,
                                               input_name='input_batch')
    quantize_static(model_input=float_path, model_output=quantized_path,
                    calibration_data_reader=onnx_q_loader)
```

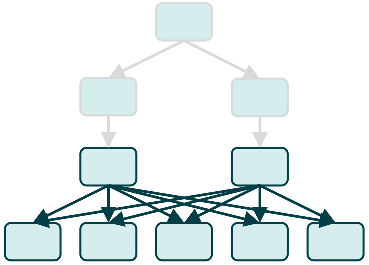
Pitfalls & Limitations

- Different backends support different operations
- Safe choices for architectures are
  - Convolution (including grouping)
  - Add
  - ReLU
  - Nearest neighbor upsampling
  - Adaptive average pooling
  - Strictly sequential graphs
- Memory layout requirements vary



C++: Efficient input pipeline

- Efficient models are not enough
- Inefficient handling of camera input can cost a lot
- Some useful optimizations
  - Resize frames early
  - Merge float conversion and normalization
  - Merge reordering and splitting of channels
  - Use buffers to hold frames

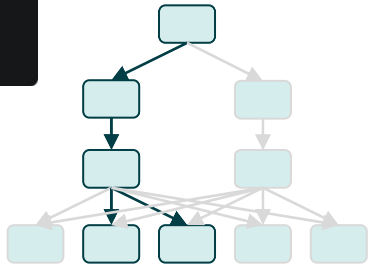


Example Use-Cases

Jetson Nano

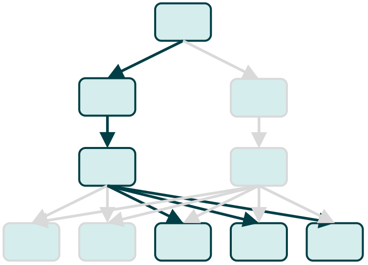
- Has images with latest PyTorch versions available
- Supports CUDA
- Can run TorchScript float models on GPU

```cpp
torch::jit::Module m_module = torch::jit::load(model_path);
m_module->to(at::kCUDA);

auto torch_tensor = torch::from_blob(input_tensor.get_data(), input_tensor.get_shape(),
                                     torch::kFloat32);
torch_tensor = torch_tensor.to(at::kCUDA);

const auto prediction = m_module->forward({torch_tensor}).toTensor().to(at::kCPU);
```



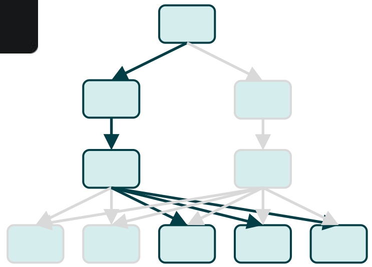
Android Phone

- PyTorch Mobile can be used directly in Java/Kotlin
- Recommended: native C++ library using LibTorch
- Available accelerators can be used with little overhead
- Quantized models on CPU are good fallback



Android Phone

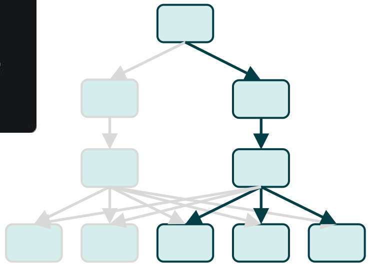
```
torch::jit::Module m_module = torch::jit::load(model_path);

auto torch_tensor = torch::from_blob(input_tensor.get_data(), input_tensor.get_shape(),
                                     torch::kFloat32); // Suffices for (quantized) models on CPU

// For models on GPU using Vulkan
torch_tensor = torch_tensor.vulkan();

// For models with float interface using NNAPI
torch_tensor = torch_tensor.contiguous(at::MemoryFormat::ChannelsLast);

const auto prediction = m_module->forward({torch_tensor}).toTensor().cpu();
```



i.MX 8M Plus

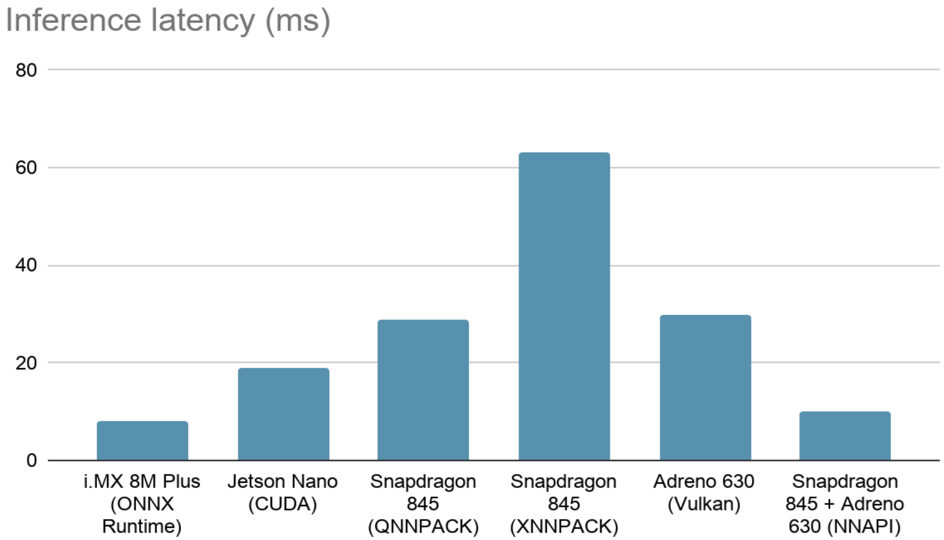
- Yocto images include PyTorch and ONNX Runtime
- PyTorch version can only access CPU
- Execution provider for ONNX Runtime to use NPU/GPU
- Convert PyTorch models to quantized ONNX models

```
Ort::SessionOptions session_options;
Ort::ThrowOnError(OrtSessionOptionsAppendExecutionProvider_VsiNpu(session_options, 0));
session_options.SetGraphOptimizationLevel(GraphOptimizationLevel::ORT_ENABLE_EXTENDED);
m_session = std::make_unique<Ort::Session>(*m_env, model_files.forward.c_str(), session_options);

const auto onnx_tensor = Ort::Value::CreateTensor<float>(*m_memory_info, input_tensor.get_data(),
                         m_input_elements, input_tensor.get_shape().data(),
                         input_tensor.get_shape().size());
auto output_tensors = m_session->Run(Ort::RunOptions{nullptr}, m_input_names.data(), &onnx_tensor,
                         1, m_output_names.data(), m_output_names.size());
```



Runtime Comparison

- Classifier with full MobileNetV2 backbone
- Hardware: i.MX 8M Plus vs. Jetson Nano vs. OnePlus 6



Conclusion

Summary

- PyTorch facilitates model development & prototyping
- Easy model architecture optimization
- Provides workflows for Arm CPU, GPU and NPU/DSP
- ONNX is a powerful alternative for inference
- Deployment to all popular hardware platforms

Resources

- PyTorch website
- TorchVision
- PyTorch Mobile
- ONNX
- ONNX Runtime
- ONNX Runtime Quantization
- PyTorch ONNX Export
- PyTorch Quantization
- PyTorch Mobile Optimizer
- PyTorch IOS GPU Workflow
- PyTorch Android GPU Workflow
- PyTorch NNAPI Workflow
- QNNPACK blog post

Example code of this talk

Talk is based on PyTorch 1.8.1 and ONNX Runtime 1.7