Group 2:
Atiya Kailany
Henry Lao
Manish Kakarla
Pradeep Gummidipundi

Exercise Explore: PyTorch Mobile

PyTorch Mobile

To minimize latency, protect privacy, and allow new interactive use cases, it's becoming more important to run machine learning models on edge devices.

The PyTorch Mobile runtime beta release enables you to go from training to deploying a model while remaining completely within the PyTorch ecosystem. It offers an end-to-end workflow for mobile devices that streamlines the research to the manufacturing process. Furthermore, using federated learning approaches, it paves the road for privacy-preserving features.

PyTorch Mobile is currently in beta and is being used in large-scale production. Once the APIs are locked down, it will be released as a stable release soon.

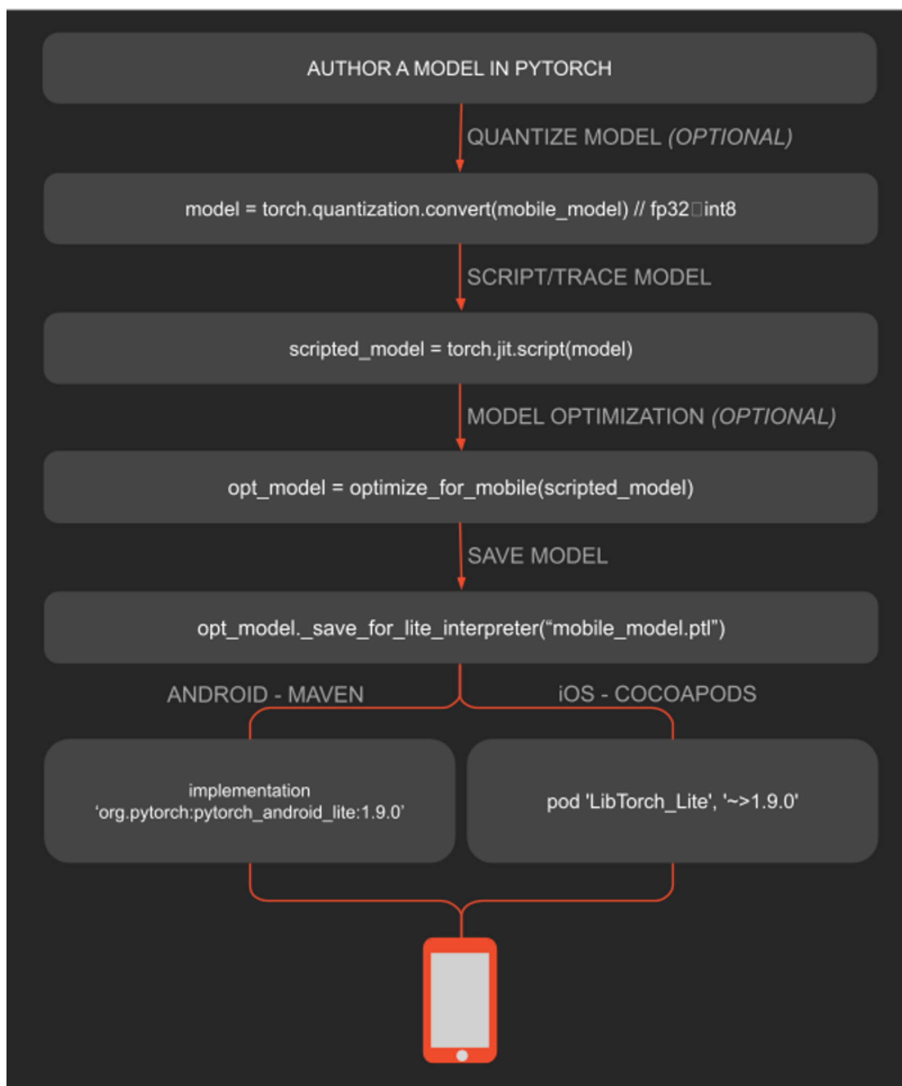
Features Offered

- Available for iOS, Android, and Linux
- Provides APIs for typical preprocessing and integration activities required for embedding machine learning into mobile apps.
- QNNPACK integration for 8-bit quantized kernels. Per-channel quantization, dynamic quantization, and other features are included.
- In Android and iOS, it provides a useful mobile translator. Additionally, based on the operators required by user programs, build level optimization and selective compilation are supported (i.e., the final binary size of the app is determined by the actual operators the app needs).
- Support for XNNPACK floating point kernel libraries for Arm CPUs.

- Integration of QNNPACK for 8-bit quantized kernels. Includes support for per-channel quantization, dynamic quantization, and more.
- Streamline model optimization via optimize_for_mobile
- Hardware backends such as GPU, DSP, and NPU will be available in Beta shortly.

Deployment Workflow

The following diagram depicts a typical pipeline from training through mobile deployment, including optional model optimization phases.



Model Preparation

Let's begin by preparing the model. If you're familiar with PyTorch, you should be able to train and save your model with ease. In the event that you don't, we'll employ a pre-trained picture categorization model (MobileNetV2). Run the following command to install it:

```
pip install torchvision
```

To serialize the model you can use python script in the root folder of the app:

```
import torch
import torchvision
from torch.utils.mobile_optimizer import optimize_for_mobile

model = torchvision.models.mobilenet_v2(pretrained=True)
model.eval()
example = torch.rand(1, 3, 224, 224)
traced_script_module = torch.jit.trace(model, example)
traced_script_module_optimized =
optimize_for_mobile(traced_script_module)
traced_script_module_optimized._save_for_lite_interpreter("app/src/ma
in/assets/model.ptl")
```

If everything goes properly, we should have our model - `model.ptl`- produced in the android application's assets folder. That will be included as an asset in the Android application and may be utilized on the device.

About Conversion to Mobile

After creating and training a model, an essential step before the model can be deployed on iOS and Android is conversion of the Python model to the JIT format **TorchScript**.

Model Quantization (Optional)

Although it is not required, it is preferable to apply compression techniques that reduce model size and accelerate inference speed. Quantization can be used to convert a model from a 32-bit floating point integer to an 8-bit integer. Currently, there are 3 variations in which quantization can be applied to a model:

- post training dynamic quantization
- post training static quantization
- quantization aware training

I. Post Training Dynamic Quantization

Run the following to apply Dynamic Quantization, which converts all the weights in a model from 32-bit floating numbers to 8-bit integers but doesn't convert the activations to int8 till just before performing the computation on the activations, simply call torch.quantization.quantize_dynamic:

```
model_dynamic_quantized = torch.quantization.quantize_dynamic(
    model, qconfig_spec={torch.nn.Linear}, dtype=torch.qint8
)
```

An important limitation of Dynamic Quantization, while it is the easiest workflow if you do not have a pre-trained quantized model ready for use, is that it currently only supports nn.Linear and nn.LSTM in qconfig_spec, meaning that you will have to use Static Quantization or Quantization Aware Training, to be discussed later, to quantize other modules such as nn.Conv2d.

II. Post Training Static Quantization

This method converts both the weights and the activations to 8-bit integers beforehand so there won't be on-the-fly conversion on the activations during the inference, as the dynamic quantization does, hence improving the performance significantly.

```
backend = "qnnpack"
model.qconfig = torch.quantization.get_default_qconfig(backend)
torch.backends.quantized.engine = backend
model_static_quantized = torch.quantization.prepare(model,
inplace=False)
model_static_quantized =
torch.quantization.convert(model_static_quantized, inplace=False)
```

III. Quantization Aware Training

This method of quanization applies the insertion of pseudo quantization to all weight and activations during model training and results in higher inference accuracy. This is typically used in CNN models.

Models can be enabled for quantization aware training by defining in the `__init__` method of the model implementation using `QuantStub` and a `DeQuantStub` to convert tensors from a floating point to quantized type and vice versa

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

To run quantization aware training, do the following:

```
model.qconfig = torch.quantization.get_default_qat_qconfig(backend)
model_qat = torch.quantization.prepare_qat(model, inplace=False)
# quantization aware training goes here
model_qat = torch.quantization.convert(model_qat.eval(),
inplace=False)
```

Conversion to TorchScript

There are two ways of converting a PyTorch model to TorchScript: using `trace` and `script`. Mixture of the two is permissible and may be necessary in some situations. More information on converting to TorchScript:

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html
- https://pytorch.org/tutorials/recipes/script_optimized.html

I. Using the *trace* Method

To use the trace method on a model, an example or dummy input for the model needs to be specified, the actual input size needs to be the same as the example input size, and the model definition cannot have control flow such as if or for. The reason for these constraints is that running `trace` on a model with an example input simply calls the model's forward method with the input and all operations executed in the model layers are recorded, creating the trace of the model.

```
import torch

dummy_input = torch.rand(1, 3, 224, 224)
torchscript_model = torch.jit.trace(model_quantized, dummy_input)
```

II. Using the *script* Method

Using this method does not ensure that trace correctly records all the possible traces when a model has some flow control**.** This can be seen in the following example:

```
import torch

class MyDecisionGate(torch.nn.Module):
    def forward(self, x):
        if x.sum() > 0:
            return x
        else:
            return -x

x = torch.rand(3, 4)
traced_cell = torch.jit.trace(MyDecisionGate(), x)
print(traced_cell.code)
```

Example output:

```
TracerWarning: Converting a tensor to a Python boolean might cause
the trace to be incorrect. We can''t record the data flow of Python
values, so this value will be treated as a constant in the future.
This means that the trace might not generalize to other inputs!

  if x.sum() > 0:
def forward(self,
    x: Tensor) -> Tensor:
  return x
```

Mobile Deployment (Android)

Reading image from Android Asset

`org.pytorch.app.MainActivity` contains all of the logic. We start by reading image.jpg and converting it to `android.graphics.` Using the Android API to create a bitmap.

```
Bitmap bitmap =
BitmapFactory.decodeStream(getAssets().open("image.jpg"));
```

Loading Mobile Module

`org.pytorch.Module` represents `torch::jit::mobile::Module` that may be loaded with a load method that specifies the serialized to file model's file path.

```
Module module = Module.load(assetFilePath(this, "model.ptl"));
```

Preparing Input

```
Tensor inputTensor = TensorImageUtils.bitmapToFloat32Tensor(bitmap,
     TensorImageUtils.TORCHVISION_NORM_MEAN_RGB,
TensorImageUtils.TORCHVISION_NORM_STD_RGB);
```

All pre-trained models require input photos that are normalized in the same way, i.e. mini-batches of 3-channel RGB images of the form (3 x H x W), with H and W of at least 224. The photos must be imported into a [0, 1] range and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0]. .225]

`org.pytorch.torchvision.TensorImageUtils` is part of `org.pytorch:pytorch_android_torchvision` library. The `TensorImageUtils#bitmapToFloat32Tensor` method creates tensors in the torchvision format using `android.graphics.Bitmap` as a source.

The shape of `inputTensor` is 1x3xHxW, where H and W are the bitmap height and width, respectively.

Run Inference

```
Tensor outputTensor =
module.forward(IValue.from(inputTensor)).toTensor();
float[] scores = outputTensor.getDataAsFloatArray();
```

`org.pytorch.Module.forward` calls the forward method of the loaded module and returns `org.pytorch.Tensor` as the result. The shape of the tensor is 1x1000.

Processing results

The data is obtained via the `org.pytorch.Tensor.getDataAsFloatArray()` function, which provides a java array of floats with scores for each image net class.

Then we only need to pick the index with the highest score and get the projected class name from `ImageNetClasses`.
All ImageNet classes are stored in the `IMAGENET CLASSES` array.

```
float maxScore = -Float.MAX_VALUE;
int maxScoreIdx = -1;
for (int i = 0; i < scores.length; i++) {
  if (scores[i] > maxScore) {
     maxScore = scores[i];
     maxScoreIdx = i;
  }
}
String className = ImageNetClasses.IMAGENET_CLASSES[maxScoreIdx];
```