PyTorch 在移动和边缘设备上的部署

叶家炜

Dec 23, 2025

边缘计算和移动 AI 的兴起源于对低延迟、隐私保护以及离线能力的迫切需求。在传统的云端 AI 部署中，数据传输带来的延迟往往难以满足实时应用场景，而将模型直接运行在设备端则能有效规避这些问题。同时，用户隐私数据无需上传云端，进一步提升了安全性。PyTorch 作为 AI 开发领域的热门框架，以其动态图和灵活性深受开发者青睐，但其在边缘部署上面临模型体积庞大、计算资源受限以及跨平台兼容性等挑战。本文旨在提供从模型训练到边缘部署的全流程指南，针对初学者和中级开发者，分享实用工具和最佳实践。读者需具备基本的 PyTorch 知识以及 Android 或 iOS 移动开发基础。

1　2. PyTorch 边缘部署生态概述

PyTorch 的边缘部署生态由一系列核心工具栈构成，这些工具共同支撑从模型导出到运行的全链路。TorchScript 是 PyTorch 原生模型序列化格式，支持 Android、iOS 和 Linux 平台，通过它可以将动态图转换为静态图以提升执行效率。PyTorch Mobile 则提供专为移动端优化的运行时，直接集成到 Android 和 iOS 应用中。ExecuTorch 作为 PyTorch 2.0 之后的下一代运行时，针对嵌入式设备设计，具有更小的二进制体积和更低的内存占用。此外，ONNX 格式允许跨框架导出，并搭配相应运行时支持多平台部署，而 TorchServe 及其移动变体则适用于服务器和边缘的服务化场景。整个部署流程可概括为训练模型、进行优化、导出格式、集成到应用、运行推理并持续调优性能，这一流程确保了从开发到生产的顺畅过渡。

2　3. 模型准备与优化

模型优化是边缘部署的基础，其中量化技术尤为关键。通过将浮点权重转换为 INT8 或 FP16 格式，可以显著减少模型大小和计算量。PyTorch 的 `torch.quantization` 模块支持动态和静态量化两种模式。以动态量化为例，它在推理时实时量化激活值，而权重预先量化。这种方法简单易用，适用于快速原型验证。以下是动态量化的示例代码：

```
import torch
import torch.quantization
model = torch.hub.load('pytorch/vision', 'resnet18', pretrained=True)
model.eval()
model.qconfig = torch.quantization.get_default_qconfig('fbgemm')
torch.quantization.prepare(model, inplace=True)
# 校准数据模拟
calib_data = torch.randn(10, 3, 224, 224)
```

```
for data in calib_data:
    model(data)
quantized_model = torch.quantization.convert(model, inplace=False)
```

这段代码首先加载预训练的 ResNet-18 模型，并设置为评估模式。然后配置量化方案，使用 fbgemm 后端适合 x86 架构。prepare 函数插入量化节点，之后通过校准数据（如随机生成的图像张量）收集统计信息，最终 convert 函数完成量化转换。量化后模型大小可减少 4 倍左右，但需注意精度损失，可通过 Top-1 准确率评估。剪枝和知识蒸馏进一步优化模型，前者移除冗余权重，后者用大模型指导小模型训练。对于 TorchScript 导出，有两种主要方法：torch.jit.trace 通过示例输入追踪计算图，适合无控制流的模型；torch.jit.script 则编译 Python 代码，支持 if-else 等逻辑，但需注解复杂函数。选择取决于模型特性。以 torch.jit.trace 导出 CNN 模型为例：

```
model = MyCNN()
model.eval()
example_input = torch.randn(1, 3, 224, 224)
traced_model = torch.jit.trace(model, example_input)
traced_model.save("model.pt")
```

这里定义自定义 CNN 模型，传入示例输入进行追踪，生成静态图并保存为 .pt 文件。常见问题包括控制流不支持，可用 script 解决；动态形状则需固定输入尺寸或使用 padding 处理。
PyTorch 2.1 引入的 ExecuTorch 进一步提升边缘性能，其优势在于支持更多算子、更小二进制和低内存占用。导出流程使用 torch.export：

```
import torch.export
model = MyModel()
example_args = (torch.randn(1, 3, 224, 224),)
exported_program = torch.export.export(model, example_args)
exported_program.save("model.ep")
```

此代码捕获模型与输入的联合表示，生成 .ep 文件，支持后续 AOT 编译，适用于资源极度受限的设备。

3 4. 平台特定部署指南

3.1 4.1 Android 部署（PyTorch Mobile）

在 Android 上部署需先搭建环境，包括 Android Studio、NDK，并通过 Gradle 添加 PyTorch Mobile AAR 依赖。集成步骤从加载 TorchScript 模型开始，使用 Module.load 从 assets 读取模型文件。随后进行输入预处理，将 Bitmap 转换为 Tensor，并执行推理。完整图像分类示例代码如下：

```
Module module = Module.load(assetFilePath(this, "model.pt"));
Tensor inputTensor = ImageUtils.bitmapToFloat32Tensor(bitmap, 224, 224, 3);
IValue inputs = IValue.from(inputTensor);
Tensor outputTensor = module.forward(IValue.from(inputs)).toTensor();
float[] scores = outputTensor.getDataAsFloatArray();
```

这段 Java 代码首先加载模型，然后利用工具函数将图像转换为 normalized Float32 Tensor（尺寸 224×224，通道 3）。`forward` 方法接收 IValue 包装的输入，返回输出 Tensor，最后提取概率分数进行分类（如 argmax 取 Top-1）。为优化性能，可启用 NNAPI 委托加速 GPU/NPU，或通过 JNI 最小化 Java-Kotlin 桥接开销。

3.2 4.2 iOS 部署（PyTorch Mobile）

iOS 部署通过 CocoaPods 集成 `LibTorch-Core`，在 Xcode 中配置后即可使用。通过 `MobileModule.loadModel` 加载模型，并处理输入 Tensor。Swift 示例代码如下：

```
let module = try MobileModule.loadModel(modelPath: modelPath)
let inputTensor = MobileTensor.fromBlob(blob: inputBlob, shape: [1, 3, 224, 224])
let output = try module.forward(input: [MobileArgument(inputTensor)]).get<
    ↪ MobileTensor>(0)
let scores = output.multiDimArray()!.data.floats
```

此代码加载模型，从 Blob 数据创建输入 Tensor（需预先从 UIImage 转换），调用 `forward` 执行推理，并从输出中提取浮点数组。性能提升可通过转换为 CoreML 格式实现：使用 `coremltools` 将 TorchScript 导出为 `.mlmodel`，集成 Metal 或 ANE 加速，推理速度可提升 2-3 倍。

3.3 4.3 嵌入式设备（Raspberry Pi / Microcontrollers）

对于 Raspberry Pi 等 Linux ARM 设备，ExecuTorch 通过 `pip install executorch` 安装，支持语音识别等任务。微控制器如 STM32 或 ESP32 受限于内存，仅支持核心算子，通过 XLA 后端编译生成的 C++ 代码运行。

4 5. 高级优化与性能调优

硬件加速是性能关键。在 Android 上，PyTorch Mobile 通过 NNAPI 委托调用 GPU 或 NPU；iOS 使用 CoreML 集成 ANE 和 Metal；边缘 NPU 如 Qualcomm 的则依赖 ExecuTorch 后端。基准测试采用 TensorFlow Lite Benchmark 工具结合 PyTorch Profiler，关注延迟、内存、功耗和 Top-1 准确率等指标。常见瓶颈包括内存爆炸，可通过设置 Batch=1 和静态形状解决；冷启动慢则用 AOT 编译预热。

5 6. 实际案例与最佳实践

在移动图像分类案例中，将 MobileNetV3 导出为 TorchScript 并部署到 Android，量化后模型大小降至 10MB，推理延迟 20ms，对比浮点版精度损失小于 1%。边缘实时目标检测则将 YOLOv5 转为 ONNX，再用 ExecuTorch 在 Jetson Nano 上运行，达到 30 FPS。最佳实践包括控制模型大小低于 50MB、推理延迟低于 30ms，使用 Git LFS 版本控制模型，并集成 Torch Hub 到 CI/CD 管道。

6 7. 挑战与未来展望

当前挑战包括算子支持不全、动态形状处理困难以及跨平台一致性问题。未来，PyTorch 2.x 通过 TorchDynamo 和 ExecuTorch 扩展生态，FBGEMM/TVM 集成深化硬件支持，联邦学习也将释放边缘潜力。

7 8. 结论与资源

PyTorch 边缘部署提供从 TorchScript 到 ExecuTorch 的完整路径，开发者可据需选择。立即实践官方 GitHub 示例仓库。进一步资源包括 pytorch.org/mobile 文档、pytorch.org/executorch 页面、github.com/pytorch/mobile 示例以及 PyTorch Forums 社区。

附录：完整代码仓库见 github.com/pytorch/android-demo。FAQ 示例：量化精度下降时，使用 QAT（量化感知训练）在训练中模拟量化误差，或增加校准数据集大小。