

AI in Mobile



卷首语

从 AlphaGo 到 AlphaZero，人工智能在短短的一年多内首先摘取了人类智慧的桂冠，然后又一骑绝尘的摆脱了对人类千年智慧和经验的积累，从 0 开始登上智能巅峰。在 ImageNet 竞赛中，2014 年算法也超过了人类的水平。这些事件也许不代表 AI 已经能够颠覆人类，但是确实标志着我们进入了人工智能时代。人工智能的兴起给我们带来了无限的遐想和可能性。同时，随着移动互联网和移动智能设备的普及，我们已经身处在移动互联网时代，移动智能设备在我们的生活中越来越重要，甚至某种程度上成为了我们的“器官”。人工智能结合移动互联网的广泛应用已经是近在眼前的新浪潮。

在这个新时代，如何让移动智能设备获得强大，稳定和低成本的智能能力是非常重要的。基于深度学习的人工智能技术显著的特点是数据和计算双密集。集中式的云端智能能够提供强大的智能能力，但是数据和计算的双重

压力在服务规模增大的时候在云端会带来巨大的瓶颈和成本，而在移动终端上数据传输的成本和时延会降低用户的体验。因此，将智能能力前置到终端设备，利用终端设备上不断增长的计算能力，将计算放到数据源头，将大大改善系统的性能和用户体验。

记得 2016 年 10 月在纽约的 Strata AI 大会上有幸与长期从事 AI 技术终端技术研究的韩松博士有过简短交流。我问大概多久 AI 芯片能够真正进入智能手机，韩松博士的判断是 3 年，但是我们已经看到 2017 年已经有搭载 AI 芯片的手机出现。AI 在移动终端的落地速度可能会远超我们的预期。

TalkingData 作为国内领先的独立移动大数据平台，我们一直关注移动智能技术的发展，同样我们也非常关注人工智能技术如何在移动终端设备落地。近两年我们在移动设备场景感知技术方面做了比较深入的研究，也发布了相关的开源项目 Myna。移动智能技术虽然前景美好，但在发展的道路上也充满了挑战。终端设备的计算能力，电池容量的限制，与云端技术架构的巨大差异，不同设备的性能差异等等问题都是很有挑战的技术问题。在这本小书中，我们长期从事移动终端智能化的王小辉，俞多，张自玉，韩广利，刘大伟，张永超和刘晓飞等同事各自对自己擅长的技术和领域贡献了非常宝贵的内容，或许能够为读者如何在工作中解决这些问题提供一些思路。本书的内容涉及到移动终端智能发展的现状，一个用场景和目前各种技术框架的现状和发展趋势，希望对关注这一新兴领域的读者有所帮助。

TalkingData 首席数据科学家 张夏天

TalkingData 首席数据科学家，全面负责移动大数据挖掘工作，包括移动应用推荐系统、移动广告优化、移动应用受众画像、移动设备用户画像、游戏数据挖掘、位置数据挖掘等工作。同时负责大数据机器学习算法的研究和实现工作。

聚焦最新技术热点 沉淀最优实践经验

[北京站]2018

北京·国际会议中心

演讲: 2018年4月20-22日 培训: 2018年4月18-19日

精彩案例 先睹为快



《Netflix的工程文化：是什么在激励着我们？》

Speaker: Katharina Probst

Netflix 工程总监



《浅谈前端交互的基础设施的建设》

Speaker: 程劭非 (寒冬)

淘宝 高级技术专家



《Apache Kafka的过去，现在，和未来》

Speaker: Jun Rao

Confluent 联合创始人



《深入Apache Spark流计算引擎：Structured Streaming》

Speaker: 朱诗雄

Databricks软件开发工程师，Apache Spark PMC和Committer



《人工智能系统中的安全风险》

Speaker: 李康

360网络安全北美研究院负责人，IoT安全研究院院长



《AI大数据时代电商攻防：AI对抗AI》

Speaker: 苏志刚

京东安全 硅谷研究中心负责人



《从C#看开放对编程语言发展的影响》

Speaker: Mads Torgersen

微软 C#编程语言Program Manager



《QUIC在手机微博中的应用实践》

Speaker: 聂永

新浪微博 技术专家



《Lavas: PWA的探索与最佳实践》

Speaker: 彭星

百度 资深前端工程师



《阿基米德微服务及治理平台》

Speaker: 张晋军

京东 基础架构部服务治理组负责人，架构师

8折 优惠报名中，立减1360元
团购享受更多优惠

访问官网获取更多前沿技术趋势

2018.qconbeijing.com

如有任何问题，欢迎咨询

电话: 15110019061, 微信: qcon-0410



ArchSummit

全球架构师峰会

2018 · 深圳站

从2012年开始算起，InfoQ已经举办了9场ArchSummit全球架构师峰会，有来自Microsoft、Google、Facebook、Twitter、LinkedIn、阿里巴巴、腾讯、百度等技术专家分享过他们的实践经验，至今累计已经为中国技术人奉上了近千场精彩演讲。

- 2017.07.07-08 深圳站
how to use sagas to maintain data consistency in a microservice architecture
——Chris Richardson, *POJOs in Action* 作者，知名微服务专家
- 2017.12.08-11 北京站
《创新是人类的自信》
——王坚博士，阿里巴巴集团技术委员会主席
- 2018.7.06-09 深圳站
限时**7折**报名中，名额有限，快快抢购。

7折报名中

名额有限，快快抢购

华南地区架构领域最有影响力的会议，届时有哪些专题和演讲，敬请扫描右方二维码浏览官网。



目录

05 移动端 AI 现状

12 移动端 AI 的应用场景

16 移动端机器学习框架 Caffe2 简介与实践

29 移动端机器学习框架 TensorFlowLite 简介与实践

36 移动端机器学习框架 SNPE 简介与实践

43 移动端机器学习框架 ncnn 简介与实践

58 AI 模型互通的新开放生态系统 ONNX 介绍与实践

69 移动端机器学习框架 MDL 简介与实践

移动端 AI 现状——正在发生的移动平台的 AI 革命

作者 王小辉

很多年以后，当我们回望 2017 年，会意识到对于移动互联网的发展来说，这一年是一个重要的里程碑。芯片制造商、移动操作系统提供商、深度学习框架社区以及移动应用开发者都开始转向 On Device AI，同时，这个趋势同样惠及于 IoT 产业的 Edge 端设备。本文就从这几个方面来解读一下这个趋势。

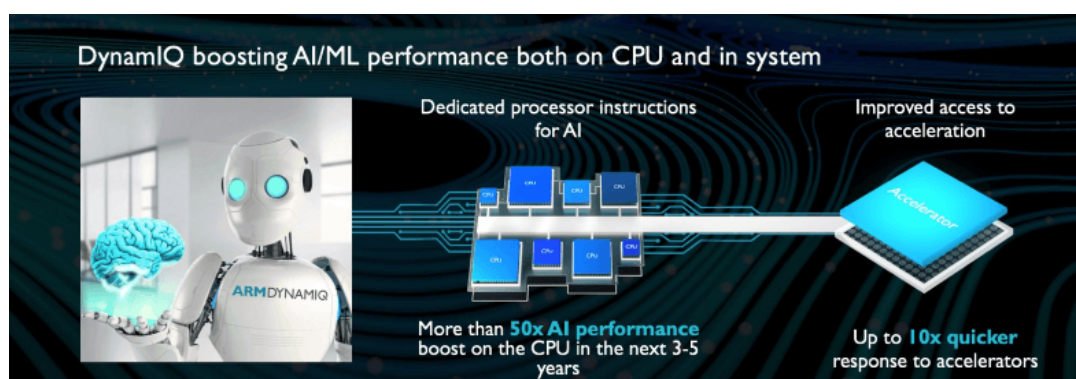
首先，为什么我们需要 On Device AI 能力呢？在 Edge 端设备上的 AI 能力可以带来这几个好处：

- 降低网络依赖，使得终端设备具备识别和决策的能力；
- 降低服务器端带宽和计算成本，因为可以将计算前置到 Edge 端设备；
- 降低延时，提高 AI 能力响应速度；
- 降低用户的移动流量成本，因为不需要上传大量原始数据到服务器端处理。

那么，Edge 端设备的硬件计算能力毕竟有限，能否支持 AI 模型的 Inference 呢？

芯片制造商

2017 年 3 月 ARM 提出了面向 AI 的新架构 [DynamIQ](#) 技术。



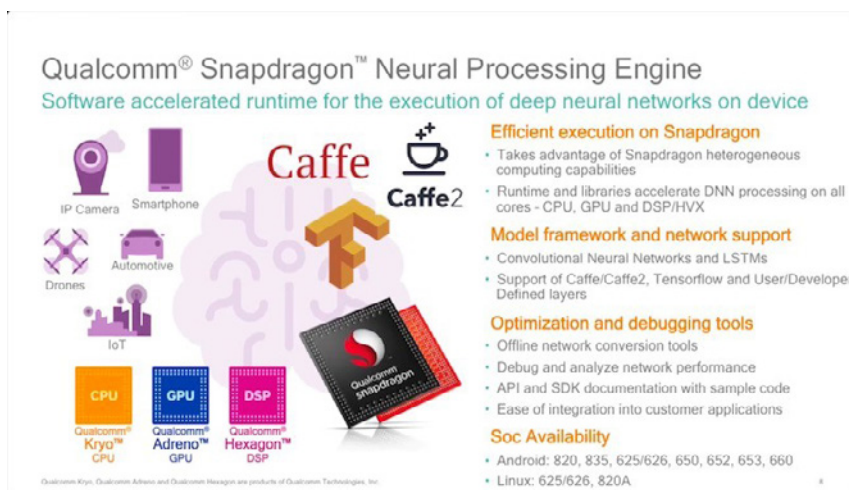
“Cortex-A CPUs that are designed based on DynamIQ technology can carry out advanced compute capabilities in Machine Learning and Artificial Intelligence. Over the next three to five years, DynamIQ-based systems will deliver up to a 50x* boost in AI performance. This is achieved through an aggressive roadmap of future DynamIQ IP, integrated with new Arm architectural instructions, microarchitectural improvements, and further software optimizations to the Arm Compute Libraries.”

ARM 表示未来 3 到 5 年内实现比基于 Cortex-A73 的设备高 50 倍的人工智能性能，最多可将 CPU 和 SoC 上特定硬件加速器的反应速度提升 10 倍。

紧接着，ARM 在 4 月份开源了支持 Cortex-A 系列 CPU 和 Mali 系列 GPU 的 Compute Library，让机器学习和深度学习算法在 ARM 平台更高效地运行。

2017 年 8 月高通对外发布了支持 Caffe/Caffe2 和 TensorFlow 的 Neural Processing Engine SDK，让深度学习可以利用 GPU 和 DSP 的计算能力。实际上，在 2017 年 4 月，Facebook 发布了针对 Edge 端设备的深度学习框架 Caffe2，其主要作者贾扬清在 F8 大会上演讲时就提到：“Android 系统上的 GPU 也类似，我们与高通合作开发了‘骁龙神经处理引擎’（SNPE），如今高通骁龙 SoC 芯片为大量的手机服务，现在是 Caffe2 的首要概念（first class

concept)，我们可以利用这些 CPU 和 DSP，从本质上提升能效和性能。”

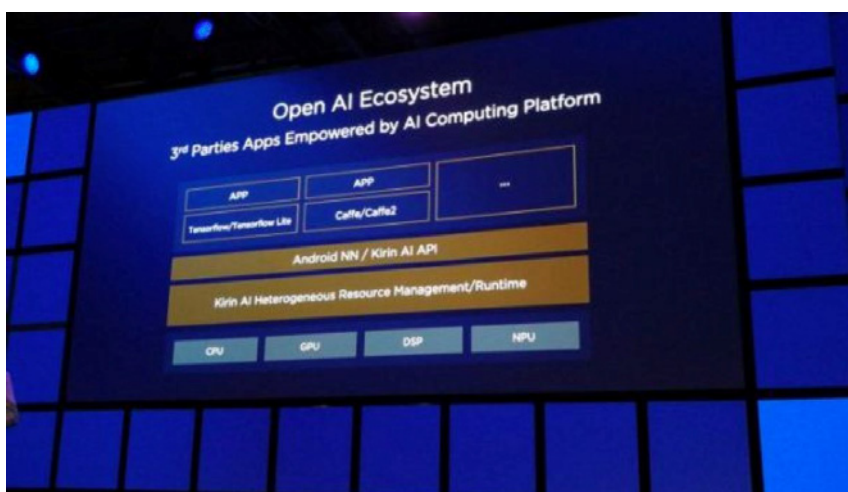


高通在 2017 年 12 月发布最新芯片骁龙 845 时提到：“In addition to the existing support for Google’s TensorFlow and Facebook’s Caffe/Caffe2 frameworks, the Snapdragon Neural Processing Engine (NPE) SDK now supports TensorFlow Lite and the new Open Neural Network Exchange (ONNX), making it easy for developers to use their framework of choice, including Caffe2, CNTK and MxNet. Snapdragon 845 also supports Google’s Android NN API.”

也就是说，SNPE 不但支持 Caffe、Caffe2、TensorFlow 和 Edge 设备端专用的 TensorFlow Lite 等深度学习框架，也支持由 Facebook 和微软发起的 ONNX (Open Neural Network Exchange)，一个 Intermediate Representation，让 Caffe2、CNTK、PyTorch 以及 MXNet 这些框架可以实现模型互通，方便芯片厂商可以针对一个标准进行硬件级别优化。

2017 年 9 月，华为发布了麒麟 970，内置中科院寒武纪 -1A NPU，可以加速神经网络在手机端的运行。

在发布旗舰设备 Mate 10 之后，华为推出了 HiAI 移动计算平台业务，践行在麒麟 970 发布会上的承诺，打造一个开放式的 AI 生态系统，让开发者可以通过这个平台为华为 Mate 10 这样搭载了最新麒麟芯片的设备上提供具有 AI



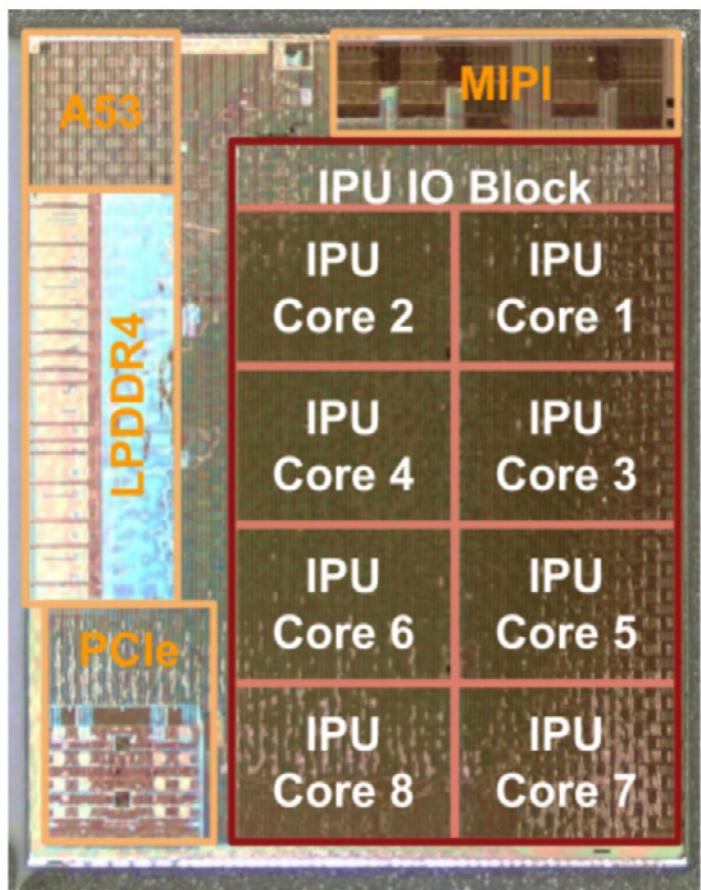
能力的应用。

2017 年 9 月，Apple 在年度产品发布会上，发布了将会搭载在 iPhone 8、iPhone 8 Plus 和 iPhone X 上的 A 11 Bionic SoC，用于支持 iOS 系统机器学习和深度学习模型的框架 CoreML 和增强现实框架 ARKit，并且升级了图形处理芯片 Metal 2，用于加速深度学习模型的 Inference。



2017 年 11 月，Google 发布了 Android 8.1 Beta 最终版本，正式支持在今年发布的新款旗舰 Pixel 2 中搭载的 IPU (Image Processing Unit)，用于加速图像处理和机器学习的 Inference: “Also, for Pixel 2 users, the Android 8.1 update on these devices enables Pixel Visual Core -- Google’s first custom-designed co-processor for image processing and

ML — through a new developer option. Once enabled, apps using Android Camera API can capture HDR+ shots through Pixel Visual Core.”



移动操作系统提供商

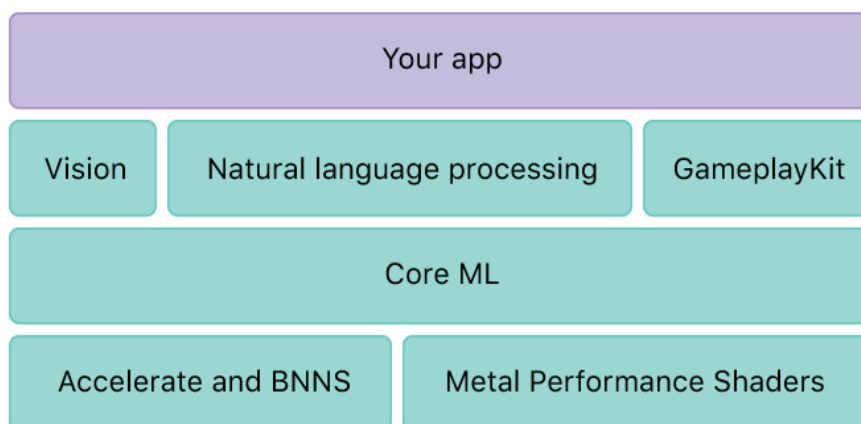
2017 年 5 月，Google 在年度 I/O 大会上，宣布会推出 TensorFlow Lite，运行在下一代 Android 系统将会新增的 Neural Network API 之上，使得开发者可以将 TensorFlow 深度学习框架创建的模型移植到移动端运行。10 月 25 日，Google 发布了 Android 8.1 Beta 开发者预览版，正式推出了 NN API：

11 月 14 日，Google 发布的 TensorFlow Lite 已经支持 NN API 了，而 DNNLibrary 是一个 GitHub 上开源的库，支持运行通过 DNN Convert Tool 转

换的 Caffe 模型。



2017 WWDC 上，Apple 发布了支持 Caffe、TensorFlow 等深度学习框架，以及 SVM、XGBoost 、 sklearn 等机器学习模型的 CoreML。12 月 5 日，Google 发布文章 Announcing Core ML support in TensorFlow Lite，宣布已经和 Apple 合作使得 CoreML 支持 TensorFlow Lite。



深度学习框架

2017 年 4 月，Facebook 宣布开源面向 Edge 端设备的深度学习框架 Caffe2，其主要作者 贾扬清在 F8 上说：“在移动端进行机器学习任务有以下好处：保护隐私，因为数据不会离开你的移动端；避免网络延迟和带宽问题；提升用户体验，比如优化 feed ranking 功能。所以我们开始着手，从底层建造一

个专门为移动端优化的机器学习框架。”



A New Lightweight, Modular, and Scalable Deep Learning Framework

2017 年 9 月，百度发布了 Mobile Deep Learning library，腾讯早些时候也开源了类似的 ncnn 框架。

2017 年以来，Google 开源了专为移动端优化的 MobileNets 模型；Face++ 提出了适合移动端的 ShuffleNet 模型。OpenCV 3.3 内置了支持多种深度学习框架的 DNN。

应用开发者

从上面的内容可以看到，从芯片制造商到移动操作系统提供商，再到深度学习框架社区，都为 On Device AI 做了很多准备，而且，从 GitHub 和国内外的开发者博客上，我们看到了非常多基于 CoreML、基于 TensorFlow 等深度学习框架的移动端应用案例，体现了开发者对这个趋势的极大热情。无论我们是哪个移动平台的开发者，都应该清晰认识到这个趋势，及时点亮自己的技能点，为用户提供更智能、更人性化的应用。

作者介绍

王小辉，目前主要负责 TalkingData 移动端基于传感器数据的场景感知应用研究和开发，以及计算机视觉技术在零售行业的解决方案。有多年的研发及架构经验，毕业于北京邮电大学。

移动端 AI 的应用场景

作者 王小辉

AI 在智能手机上有哪些应用场景呢，对于不同的开发者来说，如何确定自家的应用是否需要 AI ？如果需要，如何引入？

智能手机制造商（比如 Apple、华为等）和数以万计的应用开发者都可以在自己的 apps 中应用 AI 相关的技术，让自家的 apps 更好地服务用户的同时，提高竞争力和市场占有率。

本文将从智能手机制造商和应用开发者这两种角色出发，分别进行讨论。

智能手机制造商

2017 年，最具有明星气质的两款配备 AI 能力的手机是 iPhone X 和 华为 Mate 10，这两款手机有以下共同点：

- 搭载了专门针对 AI Inference 能力优化的硬件芯片。iPhone X 搭载了 A11 Bionic 芯片，可以加速神经网络模型运行，支持 Metal 2 的 GPU 计算能力，也可以用于深度学习模型的 Inference。
- 为开发者提供了 Inference 加速能力的 API。Apple 在最新的 iOS 11 中提供了 CoreML，通过 coremltools 将常见的传统机器学习算法模型（比如 XGBoost、SKLearn）和主流深度学习框架训练得到的模型转化

为自有格式 .mlmodel （一个整理 CoreML 格式模型的网站：coreml.store），之后该格式的模型可以打包在应用安装包里面，并在运行时加载进行 Inference；华为在 Mate 10 发布后推出了 HiAI 移动计算平台业务，进行之前发布有 NPU 的麒麟 970 芯片组时提到的 OpenAI 生态系统建设。

- 在系统应用中应用了 AI 能力。iPhone X 的 Face ID、动态表情包和 Siri 就用到了面部识别、语音识别以及 NLP 等相关 AI 能力；华为的智能相机应用、机器翻译和语音助手等也具备 Image Classification、Object Detection 等 AI 能力。

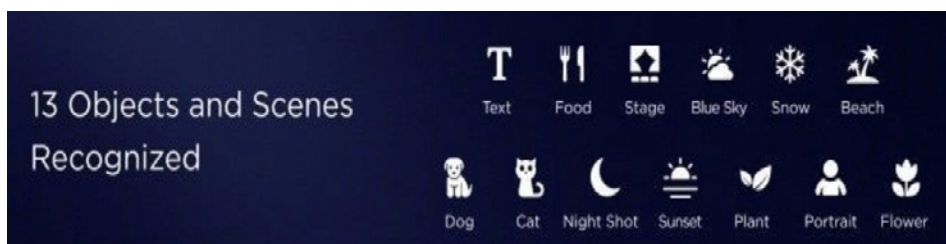
如果大家之前对这两款机型有关注和了解，对这些应该不陌生。接下来，我们整理了对于智能手机制造商来说，移动端 AI 有那些应用场景。

智能相机

- 实时识别正在被拍摄的对象和场景，进行摄像头拍摄参数的优化。



- 识别自拍或者人像拍摄，提供背景虚化。
- 拍摄时放大画面，进行自动像素增强。



- 人脸识别，结合专用摄像头阵列，提供 Face ID。
- 实时翻译：自动实时将摄像头拍摄到的文字翻译为另一种语言。

智能语音助理

- 语音识别：将语音转化为文字，这部分能力可以用于语音备忘录；如果再通过 NLP 处理，理解用户所说的话的意思，并结合上下文进行智能建议（推荐附近的餐厅、影院等）或者直接完成任务（设置定时器、订购机票以及拨打电话等）。
- 自动翻译：实时将语音翻译为另一种语言。

智能系统优化

- 基于用户的设备使用情况，智能管理设备的后台进程，减少耗电量，并且学习在合适的时候进行用户触达。
- 基于传感器数据进行用户行为识别，结合位置数据，进行 life style profiling，让设备被更好地理解自己的使用者，变得越来越个性化和智能化。

应用开发者

对于智能手机制造商来说，能够提供的系统应用毕竟是有限的，更多的用户需求需要应用开发者提供各自的解决方案。对于应用开发者来说，其实也可以提供基础类应用，比如拍照应用、智能语音助理等，如果能够借助手机制造商提供的硬件计算能力，达到比系统自带的基础类应用更好的效果，也会得到用户的欢迎。比如 Google 的输入法在 iOS 平台就比较受欢迎，该应用采用了 Google 于上半年提出的 Federated Learning，能够在提供个性化能力的同时，在线优化输入法的预测模型，并在整个过程中不上传用户的输入数据记录，保护了用户隐私。

除了基础类应用，下面这些应用场景对开发者来说也非常值得考虑：

- AR: Apple 的 ARKit 和 Google 的 ARCore 的推出意味着移动应用开

发的新趋势出现，优秀的增强现实类应用将通过更好的用户体验和交互获得用户的肯定，而为了让自家的这类应用脱颖而出，开发者除了用好 AR 框架，还需要更好的理解现实场景的能力，比如风格迁移、物体检测等都有相应的适用场景。

- 个性化内容推荐：基于用户的使用记录为用户推荐感兴趣的内容，提高用户粘性。

最后，我们还需要强调的是，无论是智能手机制造商还是应用开发者，都需要考虑对用户隐私的保护。当我们可以移动端高效运行机器学习以及深度学习模型时，有很多任务可以在移动端做，用户的数据无需上传到服务器进行处理，在保护隐私的同时，也减少了流量消耗和服务器成本。上面提到的 Google 推出的 Federated Learning 就是这样做的，Apple 也在自己的官方机器学习博客中介绍了自己的类似做法：Learning with Privacy at Scale。

作者介绍

王小辉，目前主要负责 TalkingData 移动端基于传感器数据的场景感知应用研究和开发，以及计算机视觉技术在零售行业的解决方案。有多年的研发及架构经验，毕业于北京邮电大学。

移动端机器学习框架 Caffe2 简介与实践

作者 俞多

Caffe2 简介

Caffe2 是一个轻量级、模块化、可扩展的深度学习框架，它为移动端实时计算做了很多优化，并支持大规模的分布式计算。

Caffe2 支持跨平台使用，既可以在云服务、Docker 使用，也可以安装在 Mac、Windows 或 Ubuntu 上使用。同时也支持在 Visual Studio、Android Studio、和 Xcode 中进行移动开发。

Caffe2 作为一个神经网络框架，具有以下优点：

- 支持最新的计算模型
- 分布式训练
- 高模块化
- 跨平台的支持
- 高效率

它为我们提供了模型搭建、训练和跨平台部署的能力。

深度学习应用

深度学习和神经网络可以用来处理很多问题。比如处理大数据集、实现自动

化、图像处理、数据统计和数学运算等。

Computer Vision

计算机视觉已经发展了很多年了，并且已经在一些比较先进的机器上进行应用了，比如一些制药设备。甚至还有车牌识别功能，可以自动为超速、闯红灯等行为开罚单。神经网络对计算机视觉应用有很大的提升。一些图像处理也正在应用中，比如识别图片中的内容。也有一些视频处理相关应用，用来自动场景分类和人脸识别。

Speech Recognition

苹果的 Siri 就用到了语音识别技术。它能够解析英语的多种口音，同时支持多种语言。这些能力都是通过 DNN、CNN 和其他机器学习方法实现的。

Chat Bots

聊天机器人一般是当在网站中点击“支持”链接时出现。它会根据提问者的问题自动进行回答。现在更多复杂的聊天机器人都是通过 DNN 实现，它们可以理解提问者的语义，并结合上下文给出更合适的回答，提供了优质的用户体验。

IoT

物联网也正在飞速发展，它一直伴随在我们的身边。通过智能家居可以区分来到家里的人是房主、客人或是不速之客，根据对应的环境可以产生不同的反应，如亮灯或警报灯。现在 AWS 已经有 IoT 平台并提供了这些能力。

Translation

翻译可以通过声音、文本或者图像识别进行。Caffe2 的教程中就展示了如何创建一个可以识别手写英文文本的基础神经网络，识别率超过 95%，而且速度

极快。



海关就曾经使用热成像处理技术来识别人们是否发烧，以防止疾病扩散。也可以通过 ML 或 DNN 处理医疗记录来找到其中数据的关系。

了解 Caffe2

在集成 Caffe2 之前先对它做一个简单的了解：

概要

- 选择合适的模型
- 初始化 Caffe2
- 向模型输入数据，并获得输出结果

重点对象

- `caffe2::NetDef` 是 Google Protobuf 实例，其中包含了计算图和训练好的权重值
- `caffe2::Predictor` 由 “initialization” `NetDef` 和 “predict” `NetDef` 两个网络构成的状态类，我们输入数据获取结果时使用的就是 “predict” `NetDef`

Library结构

Caffe2 由两部分组成：

- 一个核心 library，由 `Workspace`、`Blob`、`Net` 和 `Operator` 类构成
- 一个算子库，包含一系列的算子实现（比如卷积）

它是纯 C++ 实现的，且依赖下面的 library：

- Google Protobuf（轻量版本，约300kb）
- Eigen（基础线性代数子程序库），有效支持线性代数、矩阵和矢量运算、数值分析及其相关的算法。目前在ARM上 Eigen 是性能最好、运算最快的。

建议尽量使用 NNPACK，因为它在 ARM 上优化了卷积。NNPACK 是 Facebook 开源的 CPU 高性能运算库。

Caffe2 的错误和异常通常会被抛出为 `caffe2::EnforceNotMet`（继承自 `std::exception`）。

浅析 Predictor

Predictor 是一个状态类，通常是只被实例化一次来响应多次请求。

核心类 Predictor 的构造方法如下：

```
Predictor(const NetDef& init_net, const NetDef& predict_net)
```

两个 NetDef 类型的输入都是 Google Protocol Buffer 对象，表示着上面描述的两个计算图：

- `init_net` 通常会做一系列操作，把权重反序列化到 Workspace
- `predict_net` 指定了如何为每个输入数据执行计算图

构造函数中做了：

- 构建 workspace 对象
- 执行 `init_net`，申请内存空间并且设置参数的值
- 构建 `predict_net`（把 `caffe2::NetDef` 映射到一个 `caffe2::NetBase` 实例（通常是 `caffe2::SimpleNet`））

注意：如果构造函数在一台机器上失败了（抛异常），之后每台机器都会失败。在导出 NetDef 实例之前，需要先验证 Net 是否可以正确执行。

运算性能

从 2012 年起 Caffe2 就已经使用了 NEON 对 ARM CPU 进行过优化，所以 ARM CPU 可能会比 on-board GPUs 性能更好（Caffe2 的 NNPACK 在 ARM CPU 的实现在 iPhone 6s 之前的设备会比 Apple 的 MPSCNNConvolution 表现要好）。

对于卷积的实现，建议使用 NNPACK，因为它比大多数使用 `im2col/segmm` 的框架的速度要快很多，不止 2~3 倍。

设置 `OperatorDef::engine` 为 NNPACK 的示例：

```
def pick_engines(net):
    net = copy.deepcopy(net)
    for op in net.op:
        if op.type == "Conv":
            op.engine = "NNPACK"
        if op.type == "ConvTranspose":
            op.engine = "BLOCK"
    return net
```

对于一些非卷积的计算（主要是全连接层），我们还是需要使用 BLAS 库。比如在 iOS 上使用 Accelerate，在 Android 上使用 Eigen。

内存消耗

很多人担心 Caffe2 会大量消耗内存。虽然模型实例化和运行 Predictor 会消耗一些内存。但是所有的内存分配都和 Predictor 的 Workspace 实例有关，没有“static”内存分配，所以在 Predictor 实例被删除后 Workspace 的内存就会被释放。

AI Camera Demo 浅析

Caffe2 提供了 Android 平台的图像中物体检测的 Demo。打开应用后，应用会通过摄像头获取当前场景图片，并实时识别图片中的物体内容。Demo 中使用了训练好的模型，支持 1000 种物体的识别。从应用的角度看，识别物体的时间消耗越少越好，所以如果我们使用自己的模型时需要在精准度、范围、大小和运行速度之间权衡。

尝试编译并运行 AI Camera Demo

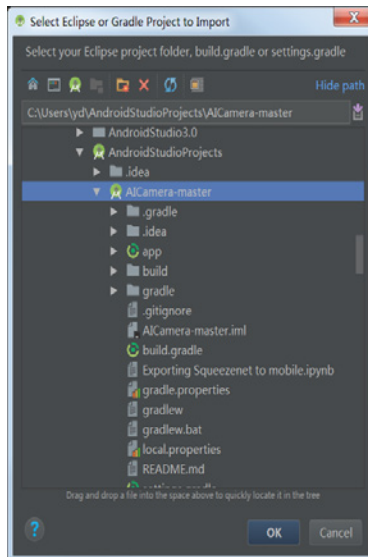
Demo Github 地址：<https://github.com/bwasti/AICamera>

1. 首先使用 git 获取项目源码：

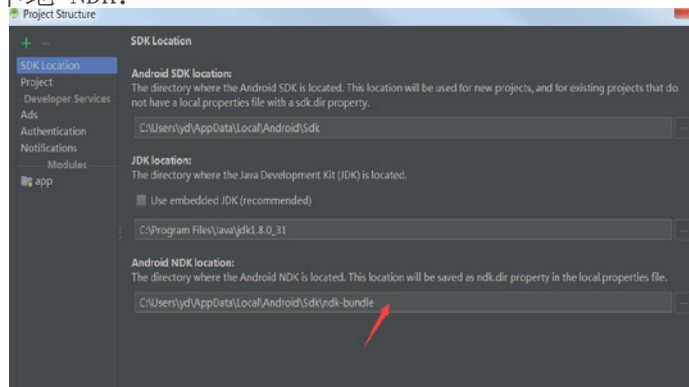
```
git clone https://github.com/bwasti/AICamera.git
git submodule init && git submodule update
cd app/libs/caffe2
git submodule init && git submodule update
```

2. 导入项目到 Android Studio 中：

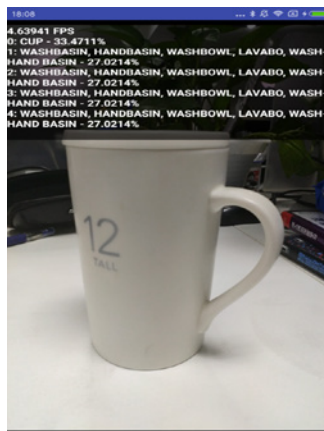
如果项目中的 gradle 插件版本和本地如果不一致，则修改为一致后再导入



3. 关联本地 NDK:

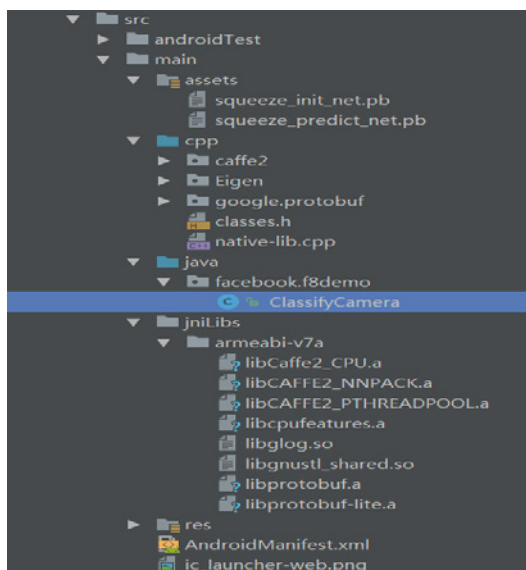


4. 编译运行后可以看到，应用正在识别拍摄到的物品:



Demo结构简析

下图为从 Android Studio 中截取的 Demo 工程结构图



- main/assets 中的文件为模型文件，为 Protobuffer 格式
- main/cpp/caffe2 为 caffe2 的C++源码库文件
- main/cpp/Eigen 为一个高层次的C++库，有效支持线性代数、矩阵和矢量运算、数值分析及其相关的算法
- main/cpp/google/protobuf 为 protobuf源码
- main/cpp/classes.h 为模型中的1000个识别种类对应的名称字符定义
- main/java/facebook/f8demo/ClassifyCamera.java 为 Demo 中唯一的一个 Activity

ClassifyCamera 中提供了两个 native 方法：

//初始化资源模型等,传入的参数为AssetManager对象，用于加载assets中的模型文件

```
public native void initCaffe2(AssetManager mgr);
```

//输入原始数据，输出分类结果

```
public native String classificationFromCaffe2(int h, int w,  
byte[] Y, byte[] U, byte[] V,  
int rowStride, int pixelStride, boolean r_hwc);
```

ClassifyCamera 在子线程中加载模型，初始化神经网络：

```

private TextView tv;
private String predictedClass = "none";
private AssetManager agr;
private boolean processing = false;
private Image image = null;
private boolean run_HWC = false;

static {
    System.loadLibrary( libname: "native-lib");
}

public native String classificationFromCaffe2(int h, int w, byte[] Y, byte[] U, byte[] V,
                                             int rowStride, int pixelStride, boolean r_hwc);

public native void initCaffe2(AssetManager agr);
private class SetupNeuralNetwork extends AsyncTask {
    @Override
    protected Void doInBackground(Void[] v) {
        try {
            initCaffe2(agr);
            predictedClass = "Neural net loaded! Inferring...";
        } catch (Exception e) {
            Log.d(TAG, "msg: 'Couldn't load neural network.'");
        }
        return null;
    }
}

```

在 Java_facebook_f8demo_ClassifyCamera_initCaffe2 中分别加载了 squeeze_init_net.pb 和 squeeze_predict_net.pb,并初始化 Predictor 对象

```

void loadNetDef(AAssetManager* agr, caffe2::NetDef* net, const char *filename) {
    AAsset* asset = AAssetManager_open(agr, filename, AASSET_MODE_BUFFER);
    assert(asset != nullptr);
    const void *data = AAsset_getBuffer(asset);
    assert(data != nullptr);
    off_t len = AAsset_getLength(asset);
    assert(len != 0);
    if (!inet->ParseFromArray(data, len)) {
        ALOG("Couldn't parse net from data.\n");
    }
    AAsset_close(asset);
}

extern "C"
void
java_facebook_f8demo_ClassifyCamera_initCaffe2(
    JNIEnv* env,
    jobject /* this */,
    jobject assetManager) {
    AAssetManager *agr = AAssetManager_fromJava(env, assetManager);
    ALOG("Attempting to load protobuf netdefs...");
    loadNetDef(agr, &_initNet, "squeeze_init_net.pb");
    loadNetDef(agr, &_predictNet, "squeeze_predict_net.pb");
    ALOG("done.");
    ALOG("Instantiating predictor...");
    _predictor = new caffe2::Predictor(_initNet, _predictNet);
    ALOG("done.");
}

```

从 CameraDevice.StateCallback 中获取到图像后，获取图片的参数，输入到 classificationFromCaffe2 方法中得到分类结果：

```

@Override
public void onImageAvailable(ImageReader reader) {
    try {
        image = reader.acquireNextImage();
        if (processing) {
            image.close();
            return;
        }
        processing = true;
        int w = image.getWidth();
        int h = image.getHeight();
        ByteBuffer Ybuffer = image.getPlanes()[0].getBuffer();
        ByteBuffer Ubuffer = image.getPlanes()[1].getBuffer();
        ByteBuffer Vbuffer = image.getPlanes()[2].getBuffer();
        // TODO: use these for proper image processing on different formats.
        int rowStride = image.getPlanes()[1].getRowStride();
        int pixelStride = image.getPlanes()[1].getPixelStride();
        byte[] Y = new byte[Ybuffer.capacity()];
        byte[] U = new byte[Ubuffer.capacity()];
        byte[] V = new byte[Vbuffer.capacity()];
        Ybuffer.get(Y);
        Ubuffer.get(U);
        Vbuffer.get(V);
        predictedClass = classificationFromCaffe2(h, w, Y, U, V,
                                                rowStride, pixelStride, run_HWC);
    }
}

```

参数说明：

- h: 图片高度
- w: 图片宽度
- y: 图片色彩y通道值
- u: 图片色彩u通道值
- v: 图片色彩v通道值
- rowStride: 一行像素头部到相邻行像素的头部的距离
- pixelStride: 一个像素头部到相邻像素的头部的距离

图片尺寸

Demo 中使用 Caffe2 处理图片时，图片需要是正方形的，所以即使从摄像头中获取到的图片不是正方形的，我们也需要把它裁剪成正方形。在 Demo 中，我们使用的图片尺寸是 227*227。

图像格式

每个相机软件或硬件对色彩处理可能不同，所以我们需要统一一个标准后再传到 Caffe2 中使用。在 Android Demo 的代码示例中我们可以看到，图片在从色彩编码使用的是 YUV，并传入了 classificationFromCaffe2 函数。但是在 classificationFromCaffe2 中，又把 YUV 格式数据转换为了 BGR。因为 Android 中 CameraPreview 中的图像数据格式为 YUV 格式，但训练模型时使用的图片是 BGR 格式，所以需要转换一下。

在 Demo 的 main/cpp/native-lib.cpp 中有详细代码，这里只给出部分代码：

```
for (auto i = 0; i < (h/2); ++i) {
    byte v_row = AF.data[0, offset + i * w];
    byte u_row = AF.data[0, offset + i * wStride];
    byte v_col = AF.data[0, offset + i * wStride];
    for (auto j = 0; j < (w/2); ++j) {
        // Convert to BGR and Y
        char y = V_row[w_offset + j];
        char u = U_row[pixelStride + (u_offset * j) / pixelStride];
        char v = V_row[pixelStride + (u_offset * j) / pixelStride];

        float b_norm = 100.00000000;
        float g_norm = 100.00000000;
        float r_norm = 100.00000000;

        auto b_i = C * 255.0 + 128.0 * j * 255.0 + 1;
        auto g_i = C * 255.0 + 128.0 * j * 255.0 + 1;
        auto r_i = C * 255.0 + 128.0 * j * 255.0 + 1;

        if (i % 2 == 0) {
            u_i = (j * 255.0 + 1) * 255.0;
            v_i = (j * 255.0 + 1) * 255.0 + 1;
            r_i = (j * 255.0 + 1) * 255.0 + 1;
        }

        B = Y + 1.002 * (v - 128);
        G = Y + 0.501 * (u - 128) + 0.7141 * (v - 128);
        R = Y + 0.7141 * (u - 128);
    }

    Input_data[2, i] = "u_norm" * (float) (float) sin(255.0 * u_i / (float) (y - 1.002 * (v - 128)));
    Input_data[3, i] = "v_norm" * (float) (float) sin(255.0 * v_i / (float) (y - 0.501 * (u - 128) - 0.7141 * (v - 128)));
    Input_data[4, i] = "r_norm" * (float) (float) sin(255.0 * r_i / (float) (y - 0.7141 * (u - 128)));
}
```

上图步骤中有几个重点：

1. 图片三通道的数据最终都合并到了一维数组 `input_data` 中
2. `b_mean`、`g_mean`、`r_mean` 是从训练数据集图片中获取的 B、G、R 三通道的平均值，在 Demo 中以硬编码方式呈现
3. 从 YUV 转换格式转换为 BGR 后，还需要减去上面提到的平均值。这样在数据整体基本不变的前提下，减小了输入数据的值，会提高接下来的运算效率。

开始计算

在获得了当前图像数据后（保存在了 `input_data` 数组中），我们就可以使用初始化好的 `Predictor` 对象来对当前图像进行计算识别了。

```
bool run(const TensorVector& inputs, TensorVector* outputs);
```

`Predictor` 的 `run` 函数需要两个参数，`inputs` 是将输入数据转换后的图像数据，`outputs` 用于保存计算后的结果。执行 `run` 函数就会开始使用模型对输入数据进行计算。

`Timer` 用于记录计算消耗时间，根据消耗的时间可以计算出 FPS 值。

```
caffe2::TensorCPU input;
if (infer_HWC) {
    input.Resize(std::vector<int>({IMG_H, IMG_W, IMG_C}));
} else {
    input.Resize(std::vector<int>({1, IMG_C, IMG_H, IMG_W}));
}
memcpy(input.mutable_data<float>(), input_data, IMG_H * IMG_W * IMG_C * sizeof(float));
caffe2::Predictor::TensorVector input_vec(&input);
caffe2::Predictor::TensorVector output_vec;
caffe2::Timer t;
t.Start();
_predictor->run(input_vec, &output_vec);
float fps = 1000/t.Milliseconds();
total_fps += fps;
avg_fps = total_fps / iters_fps;
total_fps -= avg_fps;
```

返回结果

执行 `Predictor` 的 `run` 函数后，计算的结果会保存在 `output_vec` 中。其中通过计算得到的概率会保存在 `max` 数组中，对应的识别出来的物品的 `index` 会保存在 `max_index` 中，通过 `imagenet_classes` 可以找到对应的识别出来的物品名称。Demo 中只取了识别率最高的 5 组结果。

把得到的结果和之前计算出的 FPS 值一起返回给 Android 应用层，就可以

显示在应用的界面中了。

```
constexpr int k = 5;
float max[k] = {0};
int max_index[k] = {0};
// Find the top-k results manually.
if (output_vec.capacity() > 0) {
    for (auto output : output_vec) {
        for (auto i = 0; i < output->size(); ++i) {
            for (auto j = 0; j < k; ++j) {
                if (output->template data<float>() [i] > max[j]) {
                    for (auto _j = k - 1; _j > j; --_j) {
                        max[_j + 1] = max[_j];
                        max_index[_j + 1] = max_index[_j];
                    }
                    max[j] = output->template data<float>() [i];
                    max_index[j] = i;
                    goto skip;
                }
            }
            skip;
        }
    }
}

std::ostringstream stringStream;
stringStream << avg_fps << " FPS\n";

for (auto j = 0; j < k; ++j) {
    stringStream << j << ": " << imagenet_classes[max_index[j]] << " - " << max[j] / 10 << "%\n";
}

return env->NewStringUTF(stringStream.str().c_str());
```

这就是从使用者的角度对图像识别的过程的简单分析。

识别时可能遇到的问题

- 确认输入正确的图片

使用 Caffe2 时获取合适格式的图片是至关重要的，不这样做往往会引发一些问题。有时我们会认为输入的图片都是正确的，除了检测结果精准度低外，感觉没什么问题。如果看到的识别结果像“sand dunes”、“windowpanes”、“window blinds”之类的可能是因为输入的图像是损坏的，看起来像有很多横线的电视屏幕。虽然应用在正常运行，但这不是我们想要的结果。所以 Demo 会把输入模型中的图片同时输出到屏幕中，可以直观的确认图片的格式、比例等信息是否正常。

- 需要注意我们使用的是相机哪种模式？横屏还是竖屏？

图片中的物品有可能是倾斜的，所以我们需要检查旋转并且确认我们正在使用哪种模式。这样我们才可以更准确的判断。

```
protected Matrix getMatrixFromCamera() {
    int rotation;
    if (getCameraId() == Camera.CameraInfo.CAMERA_FACING_FRONT) {
        // Undo the mirror compensation.
```



```
rotation = (360 - mOrientation) % 360;
    } else {
rotation = mOrientation;
    }

    Matrix matrix = new Matrix();
    matrix.postRotate(rotation);
    return matrix;
}
```

- 注意：如果使用的预先设置的尺寸过大，可能会超过摄像头的尺寸限制，而且还需要考虑存储的问题。

Caffe2 和 Caffe对比

Caffe 框架适用于大规模的产品用例，性能好，且有大量测试稳定的 C++ 库。Caffe 有一些从原始用例继承来的设计上的选择：传统 CNN 应用。随着新的计算模式出现，比如支持分布式计算、移动端计算、降低精度计算和其他非视觉计算，Caffe 就有了很多限制。

Caffe2 对 Caffe1.0 做了一系列的升级：

- 优化大规模分布式训练
- 支持移动端部署
- 支持新的硬件
- 已经为支持量子计算做准备
- 经过大量测试，并已经部署到生产应用中
- 模块化，可以更好地融入到业务逻辑中

总结

总而言之，Caffe2 是一个跨平台的新型工业级神经网络框架。我们在移动端、服务器端、物联网设备、嵌入式系统都能部署 Caffe2 训练的模型。我们期待在不久的将来，Caffe2 可以为通往 AI 的道路上创造新的阶梯，为我们带来不一样

的惊喜。

参考资料

<https://caffe2.ai/>

http://blog.csdn.net/melody_lu123/article/details/7772633

<http://www.cnblogs.com/raomengyang/p/4924787.html>

作者介绍

俞多，TalkingData Android SDK 研发工程师。主要负责 Android SDK 的架构设计和开发，以及自动化工程。目前业务集中在统计分析、反欺诈领域，主要探索方向为机器学习在移动端的模型训练和应用。

移动端机器学习框架 TensorFlow Lite 简介与实践

作者 张自玉

TensorFlow Lite



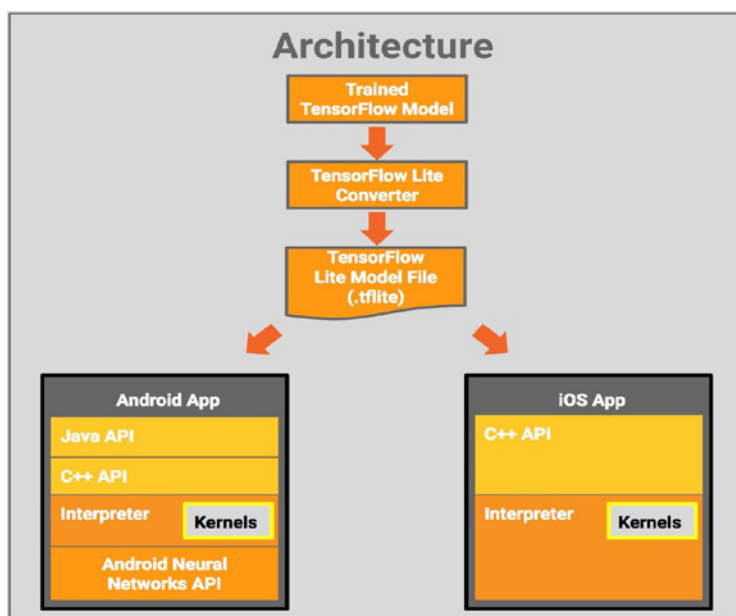
TensorFlow Lite简介

TensorFlow Lite 是 Google 在 2017 年 5 月推出的轻量级机器学习解决方案，主要针对移动设备和嵌入式设备。针对移动设备特点，TensorFlow Lite 使用了诸多技术对内核进行了定制优化，预编译激活，量化内核。TensorFlow Lite 具有以下特征：

1. 跨平台，核心代码由 C++ 编写，可以运行在不同平台上；
2. 轻量级，代码体积小，模型文件小，运行内存低；
3. 支持硬件加速。

TensorFlow Lite架构

在 server 端完成模型的训练。通过 TensorFlow Lite Converter 转化成可以在手机端运行的 .tflite 格式的模型。TensorFlow Lite 可以接入 Android 底



层的神经网络的 API 从而进行硬件加速。另外 TensorFlow 团队宣布，已经与 Apple 达成合作，可以将 TensorFlow 平台训练出来的模型转化成可被 Apple 深度学习框架 CoreML 所解释运行的格式 .mlmodel。这样一来开发者就有了更多的选择。他们既可以选择直接使用 Google 原生的 tflite 模型，也可以通过转化工具对接 Apple 的 CoreML 框架，更高效发挥 Apple 的硬件性能。CoreML 的转化工具可以从 <https://github.com/tf-coreml/tf-coreml> 去下载。

TensorFlow Lite Demo

TensorFlow Lite 提供了 iOS 和 Android 两个平台的 Demo App。这里以 iOS 平台为例，按照 TensorFlow Lite 提供的 Tutorial 完成配置。

首先安装 xcode 的工具链：

```
xcode-select --install
```

安装 automake 和 libtool：

```
brew install automake
```

```
brew install libtool
```

运行脚本。脚本的作用是下载相关依赖。主要下载 TensorFlowLite 模型：

```
tensorflow/contrib/lite/download_dependencies.sh
```

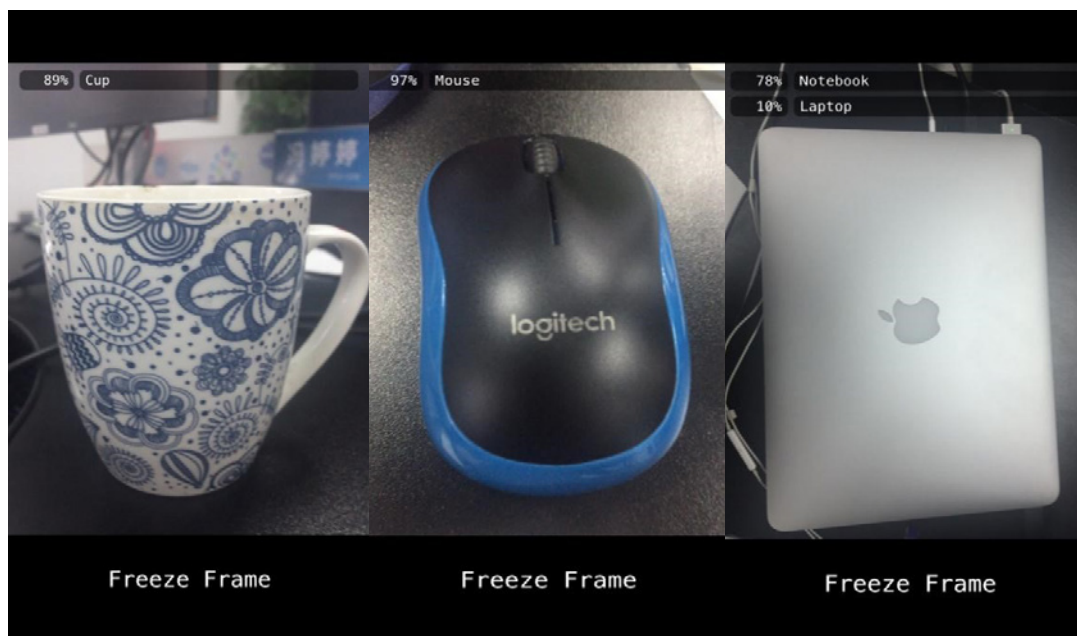
运行编译脚本，编译 TensorFlowLite 通用库：

`tensorflow/contrib/lite/build_ios_universal_lib.sh`

文件说明:

文件	类型	大小
mobilenet_quant_v1_224.tflite	模型文件	4.3M
libtensorflow-lite.a	静态库（包含多种架构 所以体积较大）	28.7M
Demo.ipa	DemoApp 的安装包	9.4M

Demo 截图:



Demo 里提供了一个由 MobileNets 训练出来的模型。Android 端的代码可以从 <https://storage.googleapis.com/download.tensorflow.org/deps/tflite/TfLiteCameraDemo.apk> 下载。iOS 端的 Demo 需要自己编译。

Demo 会把相机捕获到的每一帧转化成一个 224×224 像素的图像，这是因为训练 Mobilenet 的模型的输入都是 224×224 像素的。处理后的图像会被进一步的转化成一个 $1 \times 224 \times 224 \times 3$ 的 ByteBuffer。1 代表数量是 1， 224×224 像素的数量，3 代表每一个像素点有 3 个通道。Demo 接受单输入，输出是一个二维数组，第一维是分类索引，第二维是索引的可信程度（概率）。如上图所示。

由上表可以得知，模型文件大小为 4.3M，静态库的大小为 28.7M，这是因为

静态库是包含了所有架构的通用包，所以体积稍大。而最后打出来的安装包体积仅仅有 9.4M。在 iPhone 6 上执行实时推断。摄像头捕获帧的时间间隔大约为 4ms，说明实时推断的执行时间小于 4ms，性能非常好。

TensorFlow Lite的模型

目前 TensorFlowLite 在移动端只能解释执行 .tflite 格式的模型文件。但是一般模型训练都是在 server 端完成的。所以，想要把 server 端训练好的模型转化成可以在移动端运行的 tflite 格式，就需要对训练中的各种文件格式有一定了解。

- GraphDef (.pb)：一组代表Tensorflow训练或者计算图的protobuf数据。数据里包含了运算符、张量和变量定义。
- CheckPoint (.ckpt)：从TensorFlow图中序列化出来的变量。checkpoint里是不包含图的结构。单独的checkpoint无法被解释执行。
- FrozenGraphDef(固化权重图)：GraphDef的子类。可以通过把checkpoint里的数值带入GraphDef中得到。
- SavedModel：保存的模型。是GraphDef和CheckPoint的集合。
- TensorFlow lite model (.tflite)：一组被序列化的flatbuffer，其中包含了为例TensorFlow lite操作符和可以被TensorflowLite 解释器解释的张量。跟固化权重图很相似。

其中 .pb 文件和 .ckpt 文件都可以在 server 端生成。但是如果想在 server 端训练出来的模型可以被移动端所用，就需要将二者转化成 .tflite 格式的模型。

用 bazel 可以轻易的完成转化工作。

```
bazel build tensorflow/contrib/lite/toco:toco
```

```
bazel-bin/tensorflow/contrib/lite/toco/toco -- \
--input_file=$(pwd)/mobilenet_v1_1.0_224/frozen_graph.pb \
--input_format=TENSORFLOW_GRAPHDEF --output_format=TFLITE \
--output_file=/tmp/mobilenet_v1_1.0_224.lite --inference_
```

```

type=FLOAT \
--input_type=FLOAT --input_arrays=input \
--output_arrays=MobilenetV1/Predictions/Reshape_1 --input_
shapes=1,224,224,3

```

预训模型 (pre-trained model)

TensorFlow 官方提供了一些专门用于 TensorFlowLite 的模型。模型可以从 <https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/lite/g3doc/models.md> 下载。模型由以下训练算法得出：

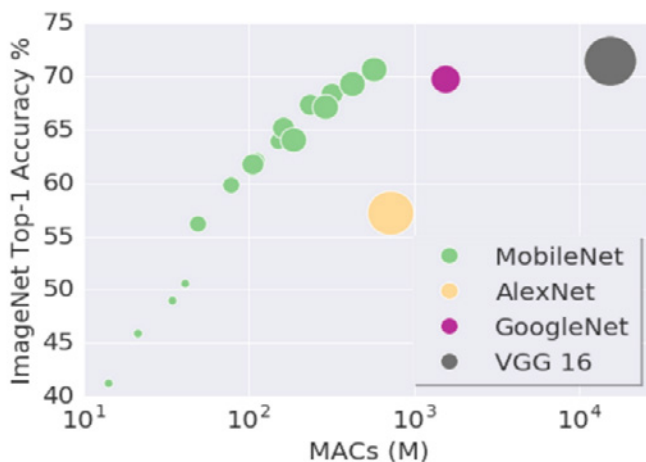
模型	类别	特性
MobileNets	计算机视觉 图像识别	低延迟，低功耗，体积轻巧
Inception-v3	计算机视觉 图像识别	高精度
On Device Smart Reply	文本处理	快速回复，目前正在 Android Wear 中有应用

迁移学习 (transfer learning)

一般来讲，计算机视觉类的模型训练通常耗时数日。但是迁移学习会把这个时间缩短到只有几个小时甚至更短。比如现在训练了一个识别猫的模型，现在需求变更为识别狗。利用迁移学习只需要把狗的训练图片作为输入，输入到已经训练好的猫类识别器里。模型会在很短的时间内完成迁移学习。

迁移学习的模型选择和首选精度

如上表格所示，Inception-v3 模型的首选精度为 78%，但是训练出的模型大小有 85M。同样的数据集在 MobileNet 上，模型大小只有 19M。下图展示了模型和首选精度的关系。



- Y轴代表的是首选精度
- X轴代表的是对应精度所代表的计算次数
- 圆形面积大小代表的是完成训练后模型的体积
- 颜色作为不同模型的区分，特别之处是这里的紫色GoogleNet其实就是Inception-v3

由此得出，想要在移动端获取实时快速响应，并且缩小模型体积的话，就需要牺牲一定的精度，来获取更好的性能。因此 MobileNet 是移动端的首选。

从TensorFlow Lite看AI-on-Device

从芯片到框架，各大软硬件厂商都在积极地布局移动端 AI。

从 AI 应用的现状来看，模型训练这些工作大多是在云端完成的，因为云端有着强大的 GPU/CPU 和内存。各种机器学习框架会充分的利用分布式、硬件加速等技术快速的完成模型的训练。从 TensorFlow Lite 可以看出 Google 在移动端 AI 的野心。目前 TensorFlow Lite 已经具备了为移动终端提供智能服务的能力。云端训练完成的模型在移动终端上可以被快速的解释执行。但是目前移动端 AI 也面临一些挑战，如：

- 移动端单机无法进行分布式训练；
- 移动端设备内存有限，部署模型受到内存限制。

但是移动端的 AI 也有很多优势：

- 移动端是数据采集的第一现场，移动端集成了多种传感器，所以传感器产生的各种数据可以第一时间被设备获取到，节省传输时间；
- 图像、语音、文本等数据大多是由手机的使用者产生。如果能在 device 上部署训练，数据的隐私会得到更好的保护；
- 移动端产生的都是个性化数据，例如行为数据，对这些数据进行“过拟合”的训练，可以产生个性化的定制服务。

正如 TensorFlow Lite 官网所说的：

“As we continue development, we hope that TensorFlow Lite will greatly simplify the developer experience of targeting a model for small devices.”

我们也希望这天早点到来。

作者介绍

张自玉，TalkingData iOS SDK 研发工程师，负责 TalkingData 统计分析平台 SDK。关注机器学习在移动端的落地。主要研究方向为 TensorFlowLite 在手机端的应用。

移动端机器学习框架 SNPE 简介与实践

译者 韩广利

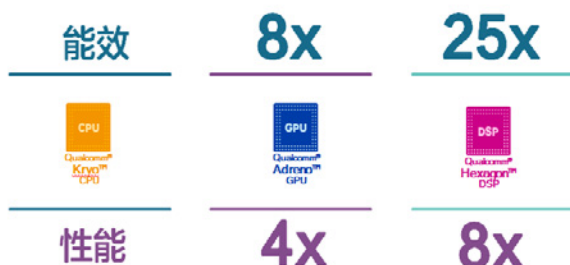
NPE SDK能够帮助开发者做什么事情？

Qualcomm 骁龙神经处理引擎 (Neural Processing Engine, NPE) SDK 能够帮助有意创建人工智能 (AI) 解决方案的开发者，在骁龙移动平台上（无论是 CPU、GPU 还是 DSP）运行通过 Caffe/Caffe2 或 TensorFlow 训练一个或多个神经网络模型，且无需连接到云端，实现边缘计算。

NPE SDK 能帮助开发者在骁龙设备上运行受过训练的神经网络并优化其性能。NPE SDK 提供了模型转换和执行工具，以及针对核的 API，利用功率和性能配置文件匹配所需的用户体验，优化和节约开发者的时间和精力。

NPE SDK 支持卷积神经网络、长短期记忆网络 (LSTM) 和定制层，处理在骁龙

Qualcomm® 骁龙™ 835 深度神经网络性能



龙移动平台上运行神经网络所需的大量繁重工作，为开发人员留出更多的时间和资源来专注于 AI 的创新应用体验。

SDK主要特性有哪些？

- 支持Android和Linux运行环境，供执行神经网络模型
- 支持利用Qualcomm Hexagon DSP、Qualcomm Adreno GPU和 Qualcomm Kryo、CPU（NPE SDK支持Qualcomm Snapdragon 820、835、625、626、650、652、653、660、630、636和450）设备必须有libOpenCL.so，以支持Qualcomm Adreno GPU），为应用提供加速
- 支持Caffe、Caffe2和TensorFlow模型
- 提供控制运行时加载、执行和调度的多个API
- 用于模型转换的桌面工具
- 用于识别性能瓶颈的性能基准测试
- 示例代码和教程
- HTML文档

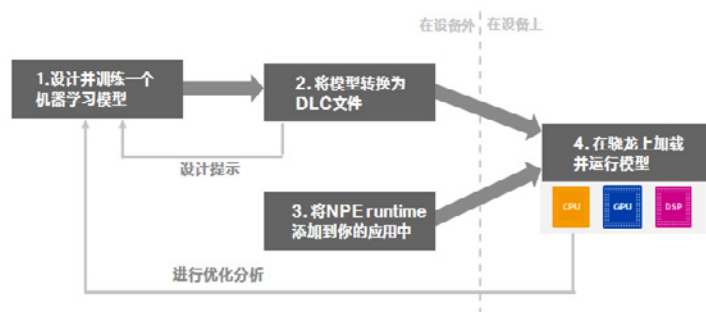
NPE SDK 适合哪些开发者？

使用骁龙 NPE SDK 开发 AI 需要满足以下几个前提，然后才可以开始创建解决方案。

- 你在一个或多个垂直领域需要运行卷积/LSTM模型，包括移动、汽车、IoT、AI、AR、无人机和机器人等
- 你了解如何设计和训练模型，或者已经有一个预训练的模型文件
- 你选择的框架是Caffe/Caffe2或TensorFlow
- 你可以使用Android编写Java应用，或者基于Android或Linux系统编写原生应用
- 你有Ubuntu 14.04开发环境
- 你有可用于测试应用程序的设备

NPE SDK使用开发流程

为了让 AI 开发者更轻松，骁龙 NPE SDK 没有另行定义网络层库；发布时就支持 Caffe/Caffe2 和 TensorFlow，开发人员可以选择使用自己熟悉的框架设计和训练网络。开发工作流程如下：



完成模型的设计和训练后，模型文件需要转换成“.dlc”（Deep Learning Container）文件，供骁龙 NPE 运行时使用。转换工具将输出转换信息，包括有关不受支持或非加速层的信息，开发者可以使用这些信息调整初始模型的设计。

搭建NPE SDK工作环境

系统环境搭建

建议在专门机器上执行以下操作，以便更好地了解 SDK 依赖项：

1. 安装 Ubuntu 14.04（官网推荐使用）

如果使用虚拟机安装，可以使用 VirtualBox 工具。虚拟机磁盘空间需要分配大些，建议分配 30G，后续 Android Studio 需要比较大的磁盘空间。

2. 安装最新版 Android Studio，地址：<https://developer.android.google.cn/studio/index.html>

通过 Android Studio 或独立安装最新版 Android SDK。

3. 安装最新版 Android NDK

通过 Android Studio SDK Manager 或独立安装。

4. 安装 Caffe，GitHub：<https://github.com/BVLC/caffe>

安装说明: <http://caffe.berkeleyvision.org/installation.html>

```
# this will build Caffe (and the pycaffe bindings) from source
- see the official instructions for more information
sudo apt-get install cmake git libatlas-base-dev libboost-
all-dev libgflags-dev libgoogle-glog-dev libhdf5-serial-
dev libleveldb-dev liblmdb-dev libopencv-dev libprotobuf-dev
libsnpappy-dev protobuf-compiler python-dev python-numpy
git clone https://github.com/BVLC/caffe.git ~/caffe; cd ~/
caffe; git reset --hard d8f79537
mkdir build; cd build; cmake ..; make all -j4; make install
```

5. 安装 TensorFlow (推荐版本 1.0, GitHub: <https://github.com/tensorflow/tensorflow>) (可选)

安装说明: <https://www.tensorflow.org/install/>

```
# this will download and install TensorFlow in a virtual
environment - see the official instructions for more information
sudo apt-get install python-pip python-dev python-virtualenv
mkdir ~/tensorflow; virtualenv -- system-site-packages ~/
tensorflow; source ~/tensorflow/bin/activate
pip install --upgrade https://storage.googleapis.com/tensorflow/
linux/cpu/tensorflow-1.0.0-cp27-none- linux_x86_64.whl
```

安装 NPE SDK

本步骤允许 NPE SDK 通过 python API 与 Caffe 和 TensorFlow 框架进行通信。
在 Ubuntu 14.04 上安装 SDK, 请执行以下操作:

1. 下载最新的骁龙 NPE SDK。

地址: <https://developer.qualcomm.com/software/snapdragon-neural-processing-engine>

将 .zip 文件解压至适当位置 (假定在 ~/snpe-sdk 文件夹中)。

2. 安装缺少的系统包:

```
# install a few more SDK dependencies, then perform a
comprehensive check
```



```

sudo apt-get install python-dev python-matplotlib python-numpy
python-protobuf python-scipy python-skimage python-sphinx wget
zip
source ~/snpe-sdk/bin/dependencies.sh # verifies that all
dependencies are installed
source ~/snpe-sdk/bin/check_python_depends.sh # verifies that
the python dependencies are installed

```

3. 在当前控制台窗口初始化 Snapdragon NPE SDK 环境。以后，每个新控制台需重复此操作：

```

# initialize the environment on the current console
• cd ~/snpe-sdk/
• export ANDROID_NDK_ROOT=~/.Android/Sdk/ndk-bundle # default
location for Android Studio, replace with yours
• source ./bin/envsetup.sh -c ~/caffe
• source ./bin/envsetup.sh -t ~/tensorflow # optional for this
guide

```

初始化过程将设置或更新 \$SNPE_ROOT, \$PATH, \$LD_LIBRARY_PATH, \$PYTHONPATH, \$CAFFE_HOME, \$TENSORFLOW_HOME, 此外，还在本地复制 Android NDK libgustl_shared.so 库，更新 Android AAR 存档文件。

下载ML Models并转换为DLC

NPE SDK 没有绑定公开的模型文件，但包含一些脚本，可用于下载一些主流模型，并将其转换为 Deep Learning Container (DLC) 格式。脚本位于 /models 文件夹中，文件夹中还包含 DLC 模型。

1. 下载并转换经预先训练的 Alexnet 示例 (Caffe 格式)：

```

cd $SNPE_ROOT
python ./models/alexnet/scripts/setup_alexnet.py -a ./temp-
assets-cache -d

```

提示：查看执行 DLC 转换的 setup_alexnet.py 脚本。您可能需要针对 Caffe 模型转换执行相同的操作。

2. 可选: 下载并转换经预先训练的“inception_v3”示例(TensorFlow 格式):

```
cd $SNPE_ROOT
python ./models/inception_v3/scripts/setup_inceptionv3.py -a ./
temp-assets-cache - d
```

提示: 查看 setup_inceptionv3.py 脚本, 此脚本还对模型进行了量化, 大小缩减了 75% (91MB → 23MB)。

构建示例Android APP

示例 Android App 的源代码演示了如何正确使用 SDK。可以从 ClassifyImageTask.java 开始。示例 Android App 结合了 Snapdragon NPE 运行环境 (/android/snpe-release.aar Android 库提供) 和上述 Caffe Alexnet 示例生成的 DLC 模型。

1. 复制运行环境和模型, 为构建 App 作好准备

```
cd $SNPE_ROOT/examples/android/image-classifiers
cp ../../../../android/snpe-release.aar ./app/libs # copies the
NPE runtime library
bash ./setup_models.sh # packages the Alexnet example (DLC,
labels, inputs) as an Android resource file
```

可选方法 1: 从 Android studio 构建 Android APK:

1. 启动 Android Studio。
2. 打开 ~/snpe-sdk/examples/android/image-classifiers 文件夹中的项目。
3. 如有的话, 接受 Android Studio 建议, 升级 构建系统组件。
4. 按下“运行应用”按钮, 构建并运行 APK。

可选方法 2: 从命令行构建 Android APK:

```
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
lib32z1
libbz2-1.0:i386 # Android SDK build dependencies on ubuntu
./gradlew assembleDebug # build the APK
```

上述命令需要将 ANDROID_HOME 和 JAVA_HOME 设置为系统中的 Android SDK 和 JRE/JDK 所在位置。

执行到这里完成后，示例 App 已经 build 完成，安装后如下所示：

使用 Snapdragon NPE SDK 制作了第一款示例应用。那么现在，可以开始创建属于自己的 AI 解决方案了！ SDK 随附文档中还有 API 文档、教程和架构详细资料。可以在浏览器中打开 /doc/html/index.html 开始学习。

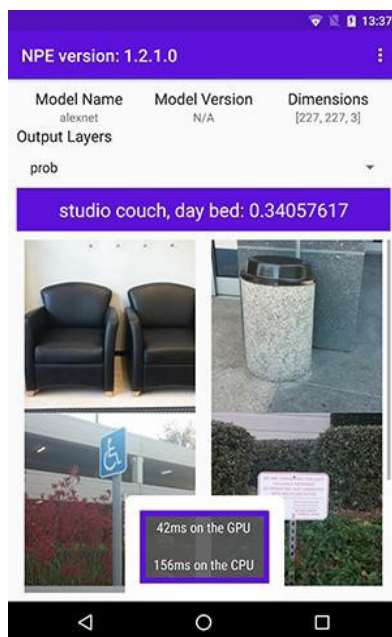
总结

NPE SDK 目前做的还不是非常完美，有些需要定制化的神经网络层可能在原生 NPE 中没有提供。但还好 SDK 提供了用户定义层（UDL）功能，通过回调函数可以自定义算子，并通过重编译 C++

代码将自定义文件编译到可执行文件中。如果开发就是使用的 C++，那比较容易实现用户定义层，但如果是运行在 Android 上，开发者需要将上层 Java 代码通过 JNI 方式来调用 NPE 原生的 C++ 编译好的 .so 文件，因为用户定义层的代码是不可能预先编译到 NPE 原生 .so 文件中的，必须重新开发 NPE 的 JNI。

作者介绍

韩广利，TalkingData Android SDK 研发工程师，主要负责应用统计分析、游戏运营分析和移动广告监测 SDK 研发工作。



移动端机器学习框架 ncnn 简介与实践

编者 刘大伟

简介

ncnn 是一个为手机端极致优化的高性能神经网络前向计算框架，也是腾讯优图实验室成立以来的第一个开源项目。ncnn 从设计之初深刻考虑手机端的部署和使用，无第三方依赖，跨平台，手机端 CPU 的速度快于目前所有已知的开源框架。基于 ncnn，开发者能够将深度学习算法轻松移植到手机端高效执行，开发出人工智能 App。ncnn 目前已在腾讯多款应用中使用，如 QQ、Qzone、微信、天天 P 图等。

腾讯优图实验室以计算机视觉见长，ncnn 的许多应用方向也都在图像方面，如人像自动美颜、照片风格化、超分辨率、物体识别。深度学习算法要在手机上落地，现成的 caffe-android-lib 依赖太多，而且手机上基本不支持 cuda，需要又快又小的前向网络实现。单纯的精简 caffe 等框架依然无法满足手机 App 对安装包大小、运算速度等的苛刻要求。ncnn 作者认为，只有全部从零开始设计才能做出适合移动端的前向网络实现，因此从最初的架构设计以手机端运行为主要原则，考虑了手机端的硬件和系统差异以及调用方式。

腾讯优图 ncnn 提供的资料显示：对比目前已知的同类框架，ncnn 是 CPU

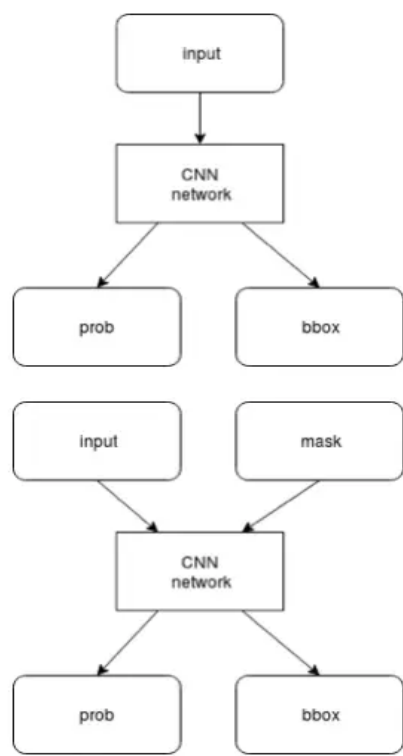
框架中最快的，安装包体积最小，跨平台兼容性中也是最好的。以苹果主推的 CoreML 为例，CoreML 是苹果主推的 iOS GPU 计算框架，速度非常快，但仅支持 iOS 11 以上的 iPhone 手机，落地受众太狭窄，非开源导致开发者无法自主扩展功能，对开源社区不友好。

ncnn 与同类框架对比。

对比	Caffe	Tensorflow	ncnn	CoreML
计算硬件	cpu	cpu	cpu	gpu
是否开源	是	是	是	否
手机计算速度	慢	慢	很快	极快
手机库大小	大	较大	小	小
手机兼容性	好	好	很好	仅支持 iOS 11

功能概述

- 支持卷积神经网络，支持多输入和多分支结构，可计算部分分支。



ncnn 支持卷积神经网络结构，以及多分支多输入的复杂网络结构，如主流的 VGG、GoogLeNet、ResNet、SqueezeNet 等。计算时可以依据需求，先计算公共部分和 prob 分支，待 prob 结果超过阈值后，再计算 bbox 分支。如果

prob 低于阈值，则可以不计算 bbox 分支，减少计算量。

- 无任何第三方库依赖，不依赖 BLAS/NNPACK 等计算框架

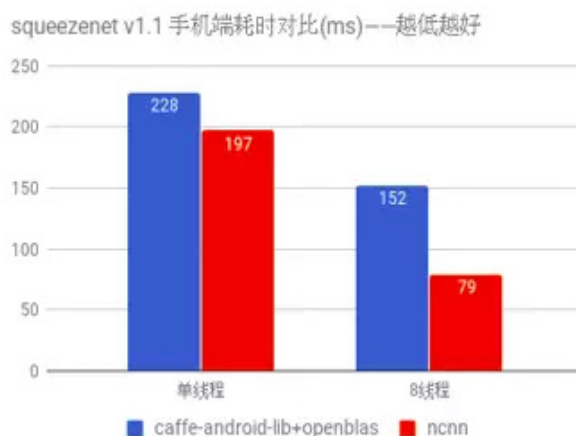
caffe-android-lib+openblas	ncnn
boost、gflags、glog、lmdb、openblas、opencv、protobuf	无

ncnn 不依赖任何第三方库，完全独立实现所有计算过程，不需要 BLAS/NNPACK 等数学计算库。

- 纯 C++ 实现，跨平台，支持 Android、iOS 等

ncnn 代码全部使用 C/C++ 实现，跨平台的 cmake 编译系统，可在已知的绝大多数平台编译运行，如 Linux、Windows、Mac OS、Android、iOS 等。由于 ncnn 不依赖第三方库，且采用 C++ 03 标准实现，只用到了 `std::vector` 和 `std::string` 两个 STL 模板，可轻松移植到其他系统和设备上。

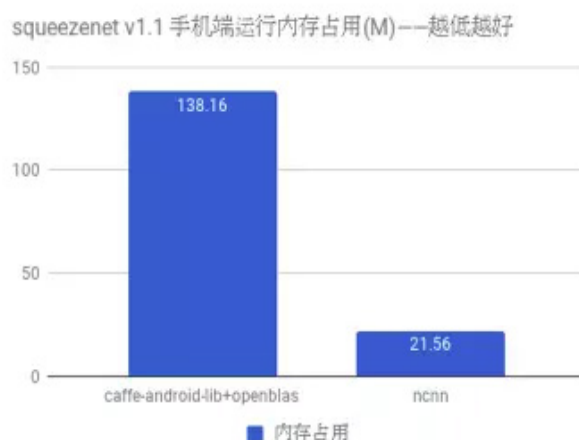
- ARM NEON 汇编级良心优化，计算速度极快



ncnn 为手机端 CPU 运行做了深度细致的优化，使用 ARM NEON 指令集实现卷积层、全连接层、池化层等大部分 CNN 关键层。对于寄存器压力较大的 armv7 架构，手工编写 neon 汇编，内存预对齐，cache 预缓存，排列流水线，充分利用一切硬件资源，防止编译器意外负优化。

测试手机为 Nexus 6p, Android 7.1.2。

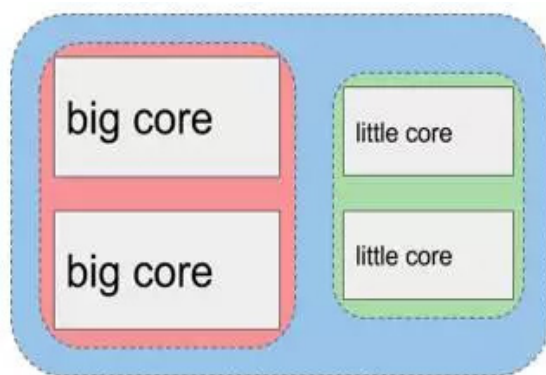
- 精细的内存管理和数据结构设计，内存占用极低



在 ncnn 设计之初已考虑到手机上内存的使用限制，在卷积层、全连接层等计算量较大的层实现中，没有采用通常框架中的 `im2col + 矩阵乘法`，因为这种方式会构造出非常大的矩阵，消耗大量内存。因此，ncnn 采用原始的滑动窗口卷积实现，并在此基础上进行优化，大幅节省了内存。在前向网络计算过程中，ncnn 可自动释放中间结果所占用的内存，进一步减少内存占用。

内存占用量使用 `top` 工具的 `RSS` 项统计，测试手机为 Nexus 6p, Android 7.1.2。

- 支持多核并行计算加速，ARM big.LITTLE CPU 调度优化



ncnn 三种模式

蓝色 = 自动

红色 = 仅大核心(高性能)

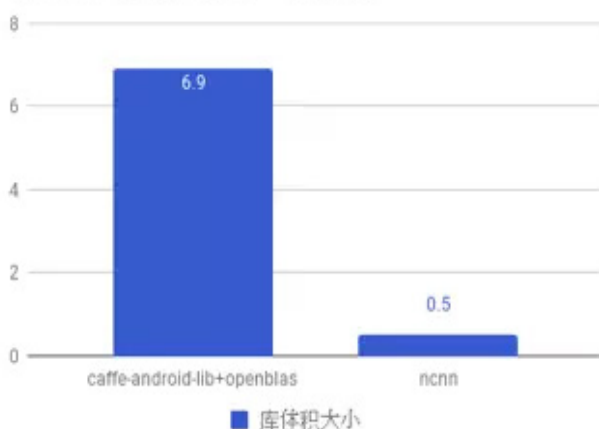
绿色 = 仅小核心(省电)

ncnn 提供了基于 OpenMP 的多核心并行计算加速，在多核心 CPU 上启用后

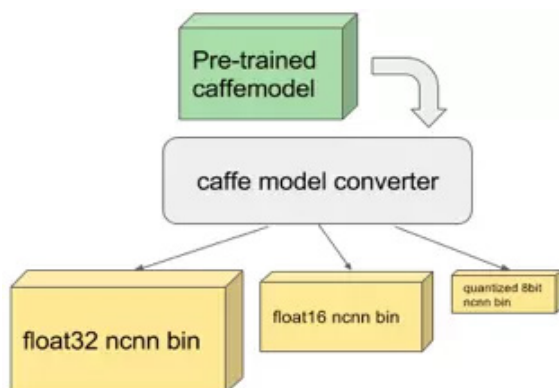
能够获得很高的加速收益。ncnn 提供线程数控制接口，可以针对每个运行实例分别调控，满足不同场景的需求。针对 ARM big.LITTLE 架构的手机 CPU，ncnn 提供了更精细的调度策略控制功能，能够指定使用大核心或者小核心，或者一起使用，获得极限性能和耗电发热之间的平衡。例如，只使用 1 个小核心，或只使用 2 个小核心，或只使用 2 个大核心，都尽在掌控之中。

- 整体库体积小于 500K，并可轻松精简到小于 300K

整体运行库二进制体积(M)——越低越好



ncnn 自身没有依赖项，且体积很小，默认编译选项下的库体积小于 500K，能够有效减轻手机 App 安装包大小负担。此外，ncnn 在编译时可自定义是否需要文件加载和字符串输出功能，还可自定义去除不需要的层实现，轻松精简到小于 300K。



ncnn 支持多种模型格式，可从caffemodel转换半精度和8bit量化模型体积更小

- 可扩展的模型设计，支持 8bit 量化和半精度浮点存储，可导入 caffe 模型

ncnn 使用自有的模型格式，模型主要存储模型中各层的权重值。ncnn 模型中含有扩展字段，用于兼容不同权重值的存储方式，如常规的单精度浮点，以及占用更小的半精度浮点和 8bit 量化数。大部分深度模型都可以采用半精度浮点减小一半的模型体积，减少 App 安装包大小和在线下载模型的耗时。ncnn 带有 caffe 模型转换器，可以转换为 ncnn 的模型格式，方便研究成果快速落地。

- 支持直接内存零拷贝引用加载网络模型

在某些特定应用场景中，如因平台层 API 只能以内存形式访问模型资源，或者希望将模型本身作为静态数据写在代码里，ncnn 提供了直接从内存引用方式加载网络模型的功能。这种加载方式不会拷贝已在内存中的模型，也无需将模型先写入实体的文件再读入，效率极高。

- 可注册自定义层实现并扩展

ncnn 提供了注册自定义层实现的扩展方式，可以将自己实现的特殊层内嵌到 ncnn 的前向计算过程中，组合出更自由的网络结构和更强大的特性。

注：只包含前向计算，因此无法进行训练，需要导入其他框架训练好的模型参数。

框架设计



框架设计与 caffe、EasyCNN 基本类似，以 Blob 存储数据，以 Layer 作为计算单元，以 Network 作为调度单元。与前 2 者稍有不同的是 ncnn 中还有一个

Extractor 的概念，Extractor 可以看做是 Network 对用户的接口。Network 一般单模型只需要一个实例，而 Extractor 可以有多个实例。这样做的好处是进行多个任务的时候可以节省内存（模型定义模型参数等不需要产生多个拷贝）。

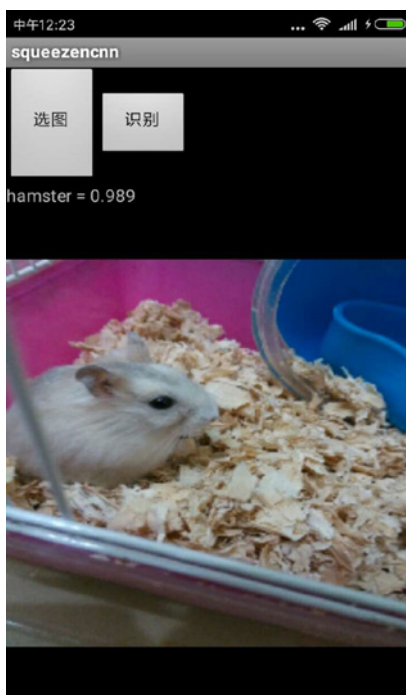
实践

ncnn功能测试

GitHub 上有可以直接运行的 Demo 工程，可以直接在 Android、iOS 上运行测试。

网址：<https://github.com/dangbo/ncnn-mobile>

运行效果如下：



在 Mac 上编译安装 ncnn

1. 下载编译源码 访问项目主页：<https://github.com/Tencent/ncnn>，使用 git 下载项目：

```
git clone git@github.com:Tencent/ncnn.git
```

2. 安装依赖环境 cmake: brew install cmake:

```
protobuf: brew install protobuf
```

3. 开始编译：

```
cd ncnn
mkdir build && cd build
cmake ..
make -j
make install
```

4. 编译成功在控制台输出：

```
Install the project...
```

```
-- Install configuration: "release"
-- Installing: /Users/liudawei/worksapace/github/ncnn/build/install/lib/libncnn.a /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/ranlib: file: /Users/liudawei/worksapace/github/ncnn/build/install/lib/libncnn.a(opencv.cpp.o) has no symbols
-- Up-to-date: /Users/liudawei/worksapace/github/ncnn/build/install/include/blob.h
-- Up-to-date: /Users/liudawei/worksapace/github/ncnn/build/install/include/cpu.h
-- Up-to-date: /Users/liudawei/worksapace/github/ncnn/build/install/include/layer.h
-- Up-to-date: /Users/liudawei/worksapace/github/ncnn/build/install/include/layer_type.h
-- Up-to-date: /Users/liudawei/worksapace/github/ncnn/build/install/include/mat.h
-- Up-to-date: /Users/liudawei/worksapace/github/ncnn/build/install/include/net.h
-- Up-to-date: /Users/liudawei/worksapace/github/ncnn/build/install/include/opencv.h
-- Up-to-date: /Users/liudawei/worksapace/github/ncnn/build/install/include/paramdict.h
-- Up-to-date: /Users/liudawei/worksapace/github/ncnn/build/install/include/layer_type_enum.h
```

```
-- Up-to-date: /Users/liudawei/worksapace/github/ncnn/build/
install/include/platform.h
```

进入 ncnn/build/tools 目录下查看生成的工具：

```
liudawei@localhost ~/worksapace/github/ncnn/build/tools master
ll
```

```
total 728
```

```
drwxr-xr-x  5 liudawei  staff   170B  12   7 17:11 CMakeFiles
```

```
-rw-r--r--  1 liudawei  staff   7.7K  12   7 16:39 Makefile
```

```
drwxr-xr-x  8 liudawei  staff   272B  12   7 17:12 caffe
```

```
-rw-r--r--  1 liudawei  staff   1.3K  12   7 16:39 cmake_install.
cmake
```

```
drwxr-xr-x  6 liudawei  staff   204B  12   7 17:12 mxnet
```

```
-rwxr-xr-x  1 liudawei  staff   351K  12   7 17:17 ncnn2mem
```

caffe模型的转换工具(caffe2ncnn)位于 caffe 目录下

```
XXXX@localhost ~/worksapace/github/ncnn/build/tools/caffe ls
```

```
CMakeFiles          caffe.pb.cc          caffe2ncnn
```

```
Makefile            caffe.pb.h           cmake_install.cmake
```

mxnet 模型的转换工具 (mxnet2ncnn) 位于 mxnet 目录下

```
XXXX@localhost ~/worksapace/github/ncnn/build/tools/mxnet ls
```

```
CMakeFiles 4Makefile cmake_install.cmake mxnet2ncnn
```

由于 ncnn 不支持模型训练，需要将其他训练好的模型做转换导入。本文中
以 caffe 为例。

- 将 caffe 模型转换为 ncnn 模型：

```
XXXX@localhost ~/worksapace/github/ncnn/build/tools/caffe ./
caffe2ncnn new_deploy.prototxt new_bvlc_alexnet.caffemodel
alexnet.param alexnet.bin
```

- 去除可见字符串 有 param 和 bin 文件其实已经可以用了，但是 param
描述文件是明文的，如果放在 App 分发出去容易被窥探到网络结构（使用
ncnn2mem 工具转换为二进制描述文件和内存模型，生成 alexnet.param.bin 和
两个静态数组的代码文件）：


```
ncnn2mem alexnet.param alexnet.bin alexnet.id.h alexnet.mem.h
```

- 加载模型

1. 直接加载 param 和 bin, 适合快速验证效果使用 :

```
ncnn::Net net;  
net.load_param("alexnet.param");  
net.load_model("alexnet.bin");
```

2. 加载二进制的 param.bin 和 bin, 没有可见字符串, 适合 App 分发模型资源 :

```
ncnn::Net net;  
net.load_param_bin("alexnet.param.bin");  
net.load_model("alexnet.bin");
```

3. 从内存引用加载网络和模型, 没有可见字符串, 模型数据全在代码里头, 没有任何外部文件 另外, Android apk 打包的资源文件读出来也是内存块 :

```
#include "alexnet.mem.h"  
ncnn::Net net;  
net.load_param(alexnet_param_bin);  
net.load_model(alexnet_bin);
```

以上三种都可以加载模型, 其中内存引用方式加载是 zero-copy 的, 所以使用 net 模型的来源内存块必须存在。

- 卸载模型

```
net.clear();
```

- 输入和输出

ncnn 用自己的数据结构 Mat 来存放输入和输出数据输入图像的数据要转换为 Mat, 依需要减去均值和乘系数:

```
#include "mat.h"  
unsigned char* rgbdata; // data pointer to RGB image pixels  
int w; // image width  
int h; // image height  
ncnn::Mat in = ncnn::Mat::from_pixels(rgbdata,
```

```
ncnn::Mat::PIXEL_RGB, w, h);
```

```
const float mean_vals[3] = {104.f, 117.f, 123.f};  
in.substract_mean_normalize(mean_vals, 0);
```

执行前向网络，获得计算结果：

```
#include "net.h"  
ncnn::Mat in;// input blob as above  
ncnn::Mat out;  
ncnn::Extractor ex = net.create_extractor();  
ex.set_light_mode(true);  
ex.input("data", in);  
ex.extract("prob", out);
```

如果是二进制的 param.bin 方式，没有可见字符串，利用 alexnet.id.h 的枚举来代替 blob 的名字：

```
#include "net.h"  
#include "alexnet.id.h"  
ncnn::Mat in;// input blob as above  
ncnn::Mat out;  
ncnn::Extractor ex = net.create_extractor();  
ex.set_light_mode(true);  
ex.input(alexnet_param_id::BLOB_data, in);  
ex.extract(alexnet_param_id::BLOB_prob, out);
```

获取 Mat 中的输出数据，Mat 内部的数据通常是三维的，c / h / w，遍历所有的 channel 获得全部分类的分数：

```
std::vector<float> scores;  
scores.resize(out.c);  
for (int j=0; j<out.c; j++)  
{  
    const float* prob = out.data + out.cstep * j;  
    scores[j] = prob[0];  
}
```

- 某些使用技巧 Extractor 有个多线程加速的开关，设置线程数能加快

计算

ex.setnumthreads(4); Mat 转换图像的时候可以顺便转换颜色和缩放大小, 这些顺带的操作也是有优化的 支持 RGB2GRAY GRAY2RGB RGB2BGR 等常用转换, 支持缩小和放大

```
#include "mat.h"
unsigned char* rgbdata;// data pointer to RGB image pixels
int w;// image width
int h;// image height
int target_width = 227;// target resized width
int target_height = 227;// target resized height
ncnn::Mat in = ncnn::Mat::from_pixels_resize(rgbdata,
ncnn::Mat::PIXEL_RGB2GRAY, w, h, target_width, target_height);
Net 有从 FILE* 文件描述加载的接口, 可以利用这点把多个网络 and 模型文
```

件合并为一个, 分发时能方便些, 内存引用就无所谓了:

```
$ cat alexnet.param.bin alexnet.bin > alexnet-all.bin
#include "net.h"
FILE* fp = fopen("alexnet-all.bin", "rb");
net.load_param_bin(fp);
net.load_bin(fp);
fclose(fp);
```

使用ncnn的example

1 . 修改 CMakeList.txt 文件, 去掉下面注释:

```
#add_subdirectory(examples)
```

2 . 使用 make -j 编译 ncnn 后, 进入 ncnn/example 目录执行:

```
./squeezenet test.jpg
```

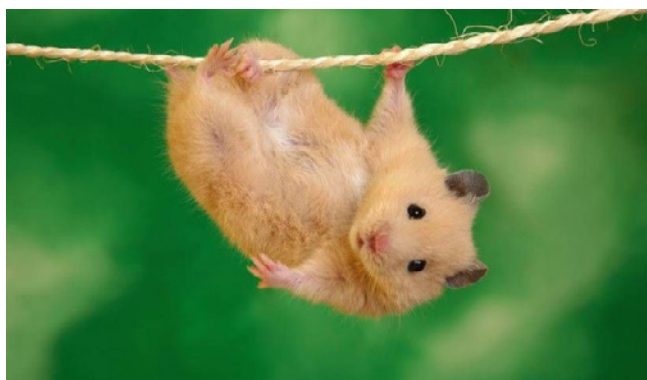
3 . 输出结果:

```
333 = 0.586699 (hamster)
```

```
186 = 0.053623 (Norwich terrier)
```

```
78 = 0.041748 (tick)
```

测试图片:



附录

Caffe 编译

1. 下载编译源码，访问项目主页：git@github.cm:BVLG/caffe.git 使用 git 下载项目

```
git clone git@github.com:BVLG/caffe.git
```

2. 安装依赖环境

使用 Homebrew 安装所需要的其它依赖，其它依赖有 gflags、snappy、glog、hdf5、lmdb、opencv3、boost、leveldb 、protobuf。

通过 pip 安装 pycaffe。

3. 开始编译

在 GitHub 上下载 Caffe 源码，地址为：<https://github.com/BVLG/caffe>，下载后在 Caffe 根目录创建 build 文件夹，将 Makefile.config.example 文件名改为 Makefile.config，修改 Makefile.config 文件。

```
$ vim Makefile.config
```

然后修改里面的内容，找到如下内容：

```
s# CPU-only switch (uncomment to build without GPU support).  
# CPU_ONLY := 1  
# CPU-only switch (uncomment to build without GPU support).  
# CPU_ONLY := 1
```

去掉注释，修改后如下：

```
# CPU-only switch (uncomment to build without GPU support).
```

```

CPU_ONLY := 1
# CPU-only switch (uncomment to build without GPU support).
CPU_ONLY := 1

```

执行 `make -j`，看到下面的内容说明安装 Caffe 成功。tools 目录下生成需要的升级工具：

```

XXXXX@localhost ~/worksapace/github/caffe/build/tools master ll
total 1536
drwxr-xr-x 16 liudawei staff 544B 12 7 17:45 CMakeFiles
-rw-r--r-- 1 liudawei staff 27K 12 7 17:45 Makefile
-rwxr-xr-x 1 liudawei staff 135K 12 7 17:49 caffe
-rw-r--r-- 1 liudawei staff 13K 12 7 17:45 cmake_install.
cmake
-rwxr-xr-x 1 liudawei staff 52K 12 7 17:49 compute_image_
mean
-rwxr-xr-x 1 liudawei staff 84K 12 7 17:49 convert_
imageset
-rwxr-xr-x 1 liudawei staff 39K 12 7 17:49 device_query
-rwxr-xr-x 1 liudawei staff 88K 12 7 17:49 extract_
features
-rwxr-xr-x 1 liudawei staff 43K 12 7 17:49 finetune_net
-rwxr-xr-x 1 liudawei staff 43K 12 7 17:49 net_speed_
benchmark
-rwxr-xr-x 1 liudawei staff 43K 12 7 17:49 test_net
-rwxr-xr-x 1 liudawei staff 43K 12 7 17:49 train_net
-rwxr-xr-x 1 liudawei staff 44K 12 7 17:49 upgrade_net_
proto_binary
-rwxr-xr-x 1 liudawei staff 44K 12 7 17:49 upgrade_net_
proto_text
-rwxr-xr-x 1 liudawei staff 44K 12 7 17:49 upgrade_solver_
proto_text

```

4. 升级 Caffe 模型

```

liudawei@localhost ~/worksapace/github/caffe/build/tools ./
upgrade_net_proto_text deploy.prototxt new_deploy.prototxt
liudawei@localhost ~/worksapace/github/caffe/build/tools ./
upgrade_net_proto_binary bvlc_alexnet.caffemodel new_bvlc_
alexnet.caffemodel

```

模型下载:

- http://dl.caffe.berkeleyvision.org/bvlc_alexnet.caffemodel
- https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet

参考内容

- <https://github.com/Tencent/ncnn>
- <https://github.com/Tencent/ncnn/wiki>
- http://blog.csdn.net/best_coder/article/details/76201275
- <http://hongbomin.com/2017/09/02/ncnn-analysis/>
- <https://mp.weixin.qq.com/s/3gTp1kqkiGwdq5olrp0vKw>

作者介绍

刘大伟, TalkingData Android SDK 研发工程师, 负责为 Android 设备应用开发者提供数据分析工具; 设计数据收集工具架构, 完成开发、测试、版本管理与维护; 主要研究 Android 系统, 分析系统数据作用并开发数据收集工具, 关注 Android 前沿技术, 包括传感器技术、数据的高效收集和处理等领域。

AI 模型互通的新开放生态系统 ONNX 介绍与实践

作者 张永超

导读



不久前，Facebook 与微软联合推出了一种开放式的神经网络切换格式——ONNX，它是一种表征深度学习模型的标准，可以实现模型在不同框架之间进行迁移。

ONNX 的全称为“Open Neural Network Exchange”，即“开放的神经网络切换”。顾名思义，该项目的目的是让不同的神经网络开发框架做到互通互用。目前，ONNX 已经得到 PyTorch、Caffe2、CNTK、MXNet 以及包括 Intel、ARM、Huawei、高通、AMD、IBM 等芯片商的支持。

按照该项目的设想，ONNX 的推出主要是为了解决当下 AI 生态系统的关键问题之一：开发框架的碎片化。现在有大量的不同框架用来构建和执行神经网络，

还有其他的机器学习系统，但是它们之间不尽相同，而且无法协同运行。



AI 的开发技术主要是深度学习神经网络，而神经网络训练和推理通常采用一种主流的深度学习框架。目前，主流的框架有如下这些：

- [TensorFlow](#) (Google)
- [Caffe/Caffe2](#) (Facebook)
- [CNTK](#) (Microsoft)
- [MXNet](#) (Amazon主导)
- [PyTorch](#) (Facebook主导)

不同的框架有不同的优势，但是框架之间的互通性并不好，甚至没有互通性。开发 AI 模型时，工程师和研究人员有很多的 AI 框架可以选择，但是并不能“一次开发，多处直接使用”。面对不同的服务平台，往往需要耗费大量的时间把模型从一个开发平台移植到另一个，让开发者苦不堪言。因此增强框架之间的互通性变的非常重要，例如：

- 情况1：某个框架的某个模型不管是准确度还是性能都非常好，但是和软件整体的架构不相符；
- 情况2：由于框架A表现很好，用它训练了一个神经网络，结果生产环境中使用的是框架B，这就意味着你可能需要将框架A训练的神经网络移植到框架B支持的格式上。

但是如果使用 ONNX，就可以消除这种尴尬，例如 ONNX 可以导出用 PyTorch 构建的训练模型，并将它与 Caffe2 结合起来用于推理（详细教程）。这对于研究阶段使用 PyTorch 构建模型，正式环境使用 Caffe2 的结构非常合适，且省去了模型移植的时间成本和人力成本等。

目前支持 ONNX 的框架如下：

Frameworks



Converters



详细支持情况，可查看 [Importing and Exporting from Frameworks](#) 中各项详细说明。

ONNX 技术概括

根据官方介绍中的内容，ONNX 为可扩展的计算图模型、内部运算器（Operator）以及标准数据类型提供了定义。每个计算图以节点列表的形式组织起来，构成一个非循环的图。节点有一个或多个的输入与输出。每个节点都是对一个运算器的调用。图还会包含协助记录其目的、作者等元数据信息。运算器在图的外部实现，但那些内置的运算器可移植到不同的框架上。每个支持 ONNX 的框架将在匹配的数据类型上提供这些运算器的实现。

概括的说就是，ONNX 是在跟踪分析 AI 框架所生成的神经网络在运行时是如何执行的，然后会利用分析的信息，创建一张可以传输的通用计算图，即符合 ONNX 标准的计算图。虽然各个框架中的计算表示法不同，但是生成的结果非常相似，因此这样的方式也是行得通的。

那么 ONNX 标准的推出，为研究者、给开发者带来的意义是什么呢？

ONNX标准的意义

- 首当其冲就是框架之间的互通性

开发者能够方便的在不同框架之间切换，为不同的任务选择最优的工具。往

往往在研发阶段需要的模型属性和产品阶段是不同的，而且不同的框架也会在特定的某个属性上有所优化，例如训练速度、对网络架构的支持、是否支持移动设备等等。如果不在研发阶段更换框架或者将研发阶段的模型进行移植，可能造成项目延迟等。ONNX 解决了这个难题，使用 ONNX 支持的 AI 框架，则可以灵活选择 AI 框架，方便地进行模型切换。

- 共享优化

芯片制造商不断地推出针对神经网络性能有所优化的硬件，如果这个优化频繁发生，那么把优化整合到各个框架是非常耗时的事。但是使用 ONNX 标准，开发者就可以直接使用优化的框架了。

这次没有Google



Google 是 TensorFlow 框架的核心主导者，而 TensorFlow 目前是业界的主流，是 GitHub 最受欢迎、生态体系健全度较高的框架。但是目前看来，TensorFlow 官方并没有支持 ONNX。但是社区已经有了非官方的 ONNX 版 TensorFlow，详情可参考 [onnx-tf](#)。

如何使用

以下是根据 ONNX 官方教程，进行的实践和遇到的具体问题。[Getting Started](#)。

ONNX 的使用总体分为两步：

1. 模型的导出: 将使用支持 ONNX 标准的 AI 框架训练的模型导出为 ONNX 格式
2. 模型的导入: 将 ONNX 格式的模型导入到支持 ONNX 的另一个 AI 框架中。

这里我选择模型导出的 AI 框架是: PyTorch, 导出的模型是 Apple Core ML 所支持的 `mlmodel` 格式。

1. ONNX 安装

本地环境:

- macOS High Sierra v10.13.1
- Python 2.7
- Xcode 9.1 command line tools
- conda 4.3.30

ONNX 的安装支持 Binaries、Docker、Source 三种方式, 由于我本地常使用 Anaconda, 因此这里我使用 Binaries 方式 进行安装:

```
conda install -c conda-forge onnx
```

安装完成后如下显示:

```
→ ~ conda install -c conda-forge onnx

Fetching package metadata .....
Solving package specifications: .

# All requested packages already installed.
# packages in environment at /Users/Robin/anaconda2:
#
onnx                                0.2.1                                py27_0    conda-forge
```

2. coremltools 安装

[coremltools](#) 的安装较为简单, 如果本地环境无误, 运行如下指令:

```
pip install coremltools
```

即可完成安装。

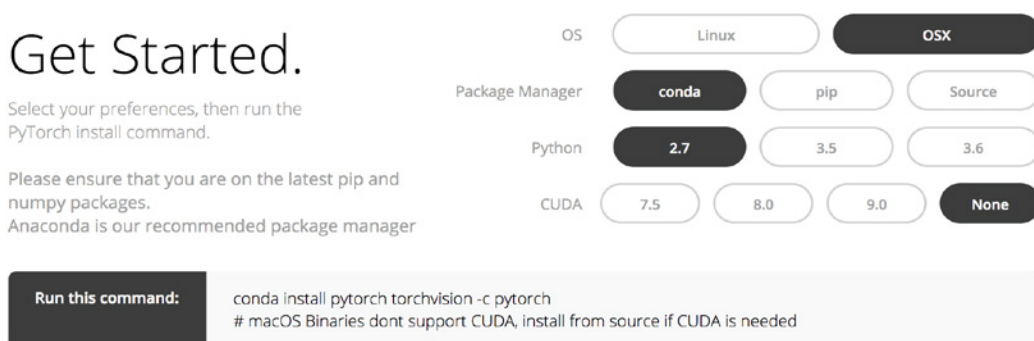
3. PyTorch 安装

PyTorch ONNX 支持的版本目前仅支持从源码 [Master](#) 分支安装, 因此如果你本机的环境中已经安装了 [PyTorch](#), 可能需要重新安装, 安装方法可见: [From](#)

[Source](#)。

需要注意的是，在进行安装时，需要将指令 `MACOSX_DEPLOYMENT_TARGET=10.9 CC=clang CXX=clang++ python setup.py install` 中的 `MACOSX_DEPLOYMENT_TARGET` 部分修改为当前你设备安装的 macOS 版本号。

截止发稿时，PyTorch 已经更新了官网版本，最新版本为 0.3.0，已经支持了 ONNX，因此你可以直接使用官网的方式安装了。【注意：如果本地已经安装过了，使用此方法安装后，需要重新安装 TorchVision (`conda install torchvision -c pytorch`)，否则部分功能可能不能使用。】



The image shows the 'Get Started' section of the PyTorch website. It includes instructions to select preferences and run the PyTorch install command. Below the instructions is a grid of buttons for OS (Linux, OSX), Package Manager (conda, pip, Source), Python (2.7, 3.5, 3.6), and CUDA (7.5, 8.0, 9.0, None). At the bottom, there is a box with the command: `conda install pytorch torchvision -c pytorch` and a note that macOS binaries don't support CUDA, so source should be installed if needed.

4. 模型算法选择

这里选择的是 PyTorch 中的一个示例模型算法 [Super-resolution](#) 进行实践。

Super-resolution（超分辨率）技术是指由低分辨率（LR，low-resolution）的图像或图像序列恢复出高分辨率（HR，high-resolution）的技术，其原理可参考[图像超分辨率技术](#)（Image Super Resolution）中的介绍。Super-resolution（超分辨率）核心思想是用时间分辨率（同一场景的多帧图像序列）换成更高的空间分辨率，实现时间分辨率向空间分辨率的转换，[论文地址](#)。

在本文中，将使用一个具有虚拟输入的小型超分辨率模型。

本文主要是讲解如何使用 ONNX 进行模型转换，对于具体的技术原理可参考文中链接。

5. 模型构建

了解了相关的软件安装和模型之后，我们开始构建 Super-resolution 模型网络，此部分内容借鉴 [PyTorch examples](#) 中的实现。

```
# Super Resolution model definition in PyTorch
import torch.nn as nn
import torch.nn.init as init

class SuperResolutionNet(nn.Module):
    def __init__(self, upscale_factor, inplace=False):
        super(SuperResolutionNet, self).__init__()
        self.relu = nn.ReLU(inplace=inplace)
        self.conv1 = nn.Conv2d(1, 64, (5, 5), (1, 1), (2, 2))
        self.conv2 = nn.Conv2d(64, 64, (3, 3), (1, 1), (1, 1))
        self.conv3 = nn.Conv2d(64, 32, (3, 3), (1, 1), (1, 1))
        self.conv4 = nn.Conv2d(32, upscale_factor ** 2, (3, 3),
                                (1, 1), (1, 1))
        self.pixel_shuffle = nn.PixelShuffle(upscale_factor)

        self._initialize_weights()

    def _initialize_weights(self):
        init.orthogonal(self.conv1.weight, init.calculate_
            gain('relu'))
        init.orthogonal(self.conv2.weight, init.calculate_
            gain('relu'))
        init.orthogonal(self.conv3.weight, init.calculate_
            gain('relu'))
        init.orthogonal(self.conv4.weight)
    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.pixel_shuffle(self.conv4(x))
        return x
```

以上模型网络构件完成后，我们就可以创建一个模型了，例如：

```
torch_model = SuperResolutionNet(upscale_factor=3)
```

此模型的结构如下：

```
SuperResolutionNet(  
    (relu): ReLU()  
    (conv1): Conv2d (1, 64, kernel_size=(5, 5), stride=(1, 1),  
padding=(2, 2))  
    (conv2): Conv2d (64, 64, kernel_size=(3, 3), stride=(1, 1),  
padding=(1, 1))  
    (conv3): Conv2d (64, 32, kernel_size=(3, 3), stride=(1, 1),  
padding=(1, 1))  
    (conv4): Conv2d (32, 9, kernel_size=(3, 3), stride=(1, 1),  
padding=(1, 1))  
    (pixel_shuffle): PixelShuffle(upscale_factor=3)  
)
```

如果你不想自己再去训练模型，可以下载已经构建好的预训练模型权重文件 [pretrained model weights](#)，进行模型的构建，例如：

```
import torch.utils.model_zoo as model_zoo  
  
# 加载预训练模型权重文件  
model_url = 'https://s3.amazonaws.com/pytorch/test_data/export/  
superres_epoch100-44c6958e.pth'  
batch_size = 1 # 这里是一个随机数（这了仅仅是为了演示使用，实际项目中需  
要调整）  
  
# 使用权重文件初始化模型  
torch_model.load_state_dict(model_zoo.load_url(model_url))  
  
# 设置训练模式为False，因为我们仅使用正向传播的方式  
torch_model.train(False)  
训练后，得到的模型结构同上。
```

6. 模型导出为ONNX格式

上面通过构建模型的过程，我们已经得到了模型，为了能够将此模型使用在 Core ML 框架中，首先需要将此模型导出为 ONNX 标准格式的模型文件。关于

PyTorch 中如何导出 ONNX 格式的模型文件，可以参考 [torch.onnx](#) 中的说明和示例。

简单的说，在 PyTorch 中要导出模型是通过跟踪其工作的方式进行的。要导出模型需要调用 `torch.onnx._export()` 函数，此函数将会记录模型中的运算符用于计算输出的轨迹，另外此函数需要一个输入张量 `x`，可以是一个图像或一个随机张量，只要它是正确的大小即可。

```
import torch.onnx
from torch.autograd import Variable
# 张量 `x` 大小确定
x = Variable(torch.randn(batch_size, 1, 224, 224), requires_grad=True)

# 导出模型
torch_out = torch.onnx._export(
    torch_model,    # 模型对象
    x, # 模型输入（或多元输入的元组）
    "onnx-model/super_resolution.onnx", # 保存文件的路径（或文件对象）
    export_params=True) # 是否存储参数（True: 将训练过的参数权重存储在模型文件中；False: 不存储）
```

`torch_out` 是执行模型后的输出。通常情况下可以忽略这个输出，但是在这里我们将使用它来验证在 Core ML 中运行导出的模型是否和此值具有计算相同的值。

7. 转换ONNX格式到Core ML支持的mlmodel格式

6.1 安装转换工具onnx-coreml

首先我们要安装能够将 ONNX 标准格式的模型转换为 Apple Core ML 格式的工具 `onnx-coreml`。安装指令如下：

```
pip install onnx-coreml
```

安装完成后，就可以使用此工具将 ONNX 标准格式的模型转换为 Apple Core ML 格式了。

6.2 加载ONNX格式模型文件

在开始之前，需要先使用 ONNX 加载之前导出的 ONNX 格式模型文件到对象中：

```
import onnx
model = onnx.load('super_resolution.onnx')
```

6.3 转换模型到CoreML

由于我们安装了 ONNX 到 Core ML 的转换工具，因此我们可以直接使用其中的 `convert()` 函数转换模型：

```
import onnx_coreml
cml = onnx_coreml.convert(model)
print type(cml)
cml.save('coreml-model/super_resolution.mlmodel')
```

这里的 `cml` 是格式转换后的 `coreml` `model` 对象，调用其 `save()` 方法即可将转换后的对象存储文件，即得到 Core ML 支持的模型文件。

有了这个 `mlmodel` 格式的模型文件后，我们就可以将其应用在 Core ML 中进行推理了，这里不再进行此部分的描述，具体使用方法可参考 Apple 官方给出的 `mlmodel` 模型文件相关的使用示例，[Integrating a Core ML Model into Your App](#)。

总结

目前，ONNX 所支持的 AI 框架和工具还较少，整个项目还处于初级阶段，但是已经能够看出，ONNX 的推出无疑是对整个 AI 研究和开发领域极大的福祉。不久的将来，当大多数的 AI 框架和工具都支持 ONNX 标准的时候，就不会发生为了选择生产环境的 AI 框架而重新学习、为了能够生产环境应用模型而耗时移植模型等等类似的事件了。减少了开发人员消耗在移植过程中的时间，增加了钻研算法和开发更令人兴奋的 AI 应用的时间，相信每个开发者和研究者都盼望着这个时期的到来。

参考资料

- [Facebook and Microsoft introduce new open ecosystem for](#)

[interchangeable AI frameworks](#)

- [Microsoft and Facebook create open ecosystem for AI model interoperability](#)
- [ONNX](#)
- [Core ML](#)
- [PyTorch](#)
- [Announcing ONNX Support for Apache MXNet](#)

作者介绍

张永超，TalkingData iOS SDK 研发工程师，六年 iOS 技术研发经验，熟悉 Objective-C、Swift 等语言，目前正关注机器学习（机器学习移动化、移动端机器学习利用）以及移动端数据安全（网络数据、本地存储）。

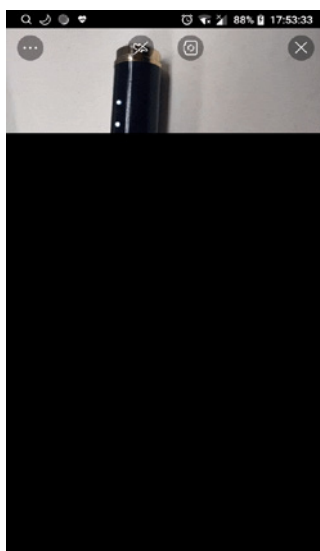
移动端机器学习框架 MDL 简介与实践

译者 刘晓飞

Mobile-deep-learning (MDL)

MDL 是百度研发的可以部署在移动端的基于卷积神经网络实现的移动端框架，可以应用在图像识别领域。

具体应用：在手机百度 App 中，用户只需要点击自动拍开关，将手机对准物体，当手停稳的时候，它会自动找到物体并进行框选，无需拍照就可以发起图像的搜索功能。



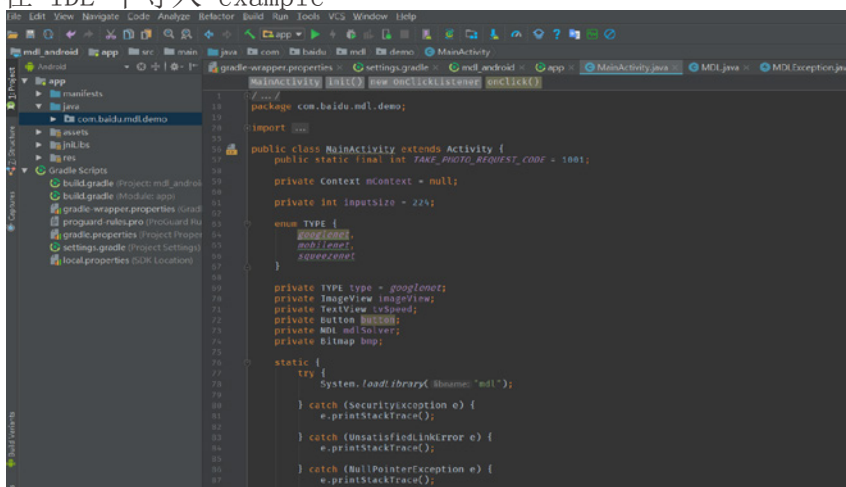
初识 MDL

运行示例程序

1. clone 项目代码, <https://github.com/baidu/mobile-deep-learning>

名称	修改日期	类型	大小
.git	2017/12/18 14:47	文件夹	
android-cmake	2017/12/18 14:47	文件夹	
examples	2017/12/18 14:47	文件夹	
include	2017/12/18 14:47	文件夹	
iOS	2017/12/18 14:47	文件夹	
ios-cmake	2017/12/18 14:47	文件夹	
scripts	2017/12/18 14:47	文件夹	
src	2017/12/18 14:47	文件夹	
test	2017/12/18 14:47	文件夹	
third-party	2017/12/18 14:47	文件夹	
tools	2017/12/18 14:47	文件夹	
.gitignore	2017/12/18 14:47	文本文档	1 KB
android_showcase.gif	2017/12/18 14:47	GIF 文件	2,342 KB
baidu_showcase.gif	2017/12/18 14:47	GIF 文件	7,321 KB
build.sh	2017/12/18 14:47	Shell Script	5 KB
CMakeLists.txt	2017/12/18 14:47	文本文档	2 KB
CONTRIBUTING.md	2017/12/18 14:47	MD 文件	1 KB
Help for Mac.md	2017/12/18 14:47	MD 文件	3 KB
LICENSE	2017/12/18 14:47	文件	2 KB
README.md	2017/12/18 14:47	MD 文件	7 KB

2. 在 IDE 中导入 example



3. 运行



开发要求

1. 安装 NDK
2. 安装 Cmake
3. 安装 protocol buffers

使用 MDL 库

1. 在 mac/linux 上执行测试

```
# mac or linux:
./build.sh mac
cd build/release/x86/build
./mdlTest
```

2. 在项目中使用 mdl

开发

1. 编译 MDL 源码 (Android)

```
# android:
# prerequisite: install ndk from google
./build.sh android
cd build/release/armv-v7a/build
./deploy_android.sh
adb shell
cd /data/local/tmp
./mdlTest
```

2. iOS

```
# ios:
# prerequisite: install xcode from apple
./build.sh ios

copy ./build/release/ios/build/libmdl-static.a to your iOS
project
```

模型转换

MDL 需要与之兼容的模型才能使用，可以使用 MDL 提供的脚本将其他深度学习工具训练的模型转换为 MDL 模型。推荐使用 PaddlePaddle 模型。

1. 将 PaddlePaddle 模型转换成 MDL 模式

```
# Environmental requirements
# paddlepaddle
cd tools/python
python paddle2mdl.py
```

2. 将 caffe model 模型转换成 MDL 模式

```
#Convert model.prototxt and model.caffemodel to model.min.json
and data.min.bin that mdl use
```

```
./build.sh mac
cd ./build/release/x86/tools/build
```

```
# copy your model.prototxt and model.caffemodel to this path
```

```
./caffe2mdl model.prototxt model.caffemodel
```

```
# the third para is optional, if you want to test the model
produced by this script, provide color value array of an image
as the third parameter ,like this:
```

```
./caffe2mdl model.prototxt model.caffemodel data
```

```
# the color value should in order of rgb,and transformed
according to the model.
```

```
# then you will get a new data.min.bin with test data inside
```

```
# after this command, model.min.json data.min.bin will be
created in current
```

```
# some difference step you need to do if you convert caffe
model to iOS GPU format
```

```
# see this:
```

```
open iOS/convert/iOSConvertREADME.md
```


Android Sample 分析

下面以 Android 平台为例分析 MDL 在移动端平台上面的工作

1. 在项目中导入 libmdl.so 库
2. 初始化 mdl, 加载 so 库, 设置线程数量

```
private void initMDL() {
    String assetPath = "mdl_demo";
    String sdcardPath = Environment.getExternalStorageDirectory()
        + File.separator + assetPath;
    copyFilesFromAssets(this, assetPath, sdcardPath);
    mdlSolver = new MDL();
    try {
        String jsonPath = sdcardPath + File.separator + type.name()
            + File.separator + "model.min.json";
        String weightsPath = sdcardPath + File.separator + type.name()
            + File.separator + "data.min.bin";
        Log.d("mdl", "mdl load " + jsonPath + "weightpath " + weightsPath);
        mdlSolver.load(jsonPath, weightsPath);
        if (type == mobilenet) {
            mdlSolver.setThreadNum(1);
        } else {
            mdlSolver.setThreadNum(3);
        }

    } catch (MDLException e) {
        e.printStackTrace();
    }
}
```

3. 拍摄照片

```
Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
// save pic in sdcard
Uri imageUri = Uri.fromFile(getTempImage());
intent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri);
startActivityForResult(intent, TAKE_PHOTO_REQUEST_CODE);
```

4. 图像处理，包括设置采样率、图像缩放

5. 图像预测

```
mdlSolver.predictImage(inputData);
```

6. 在 MDL 库中，predictImage 方法是进行图像预测的 JNI 方法。方法声明如下：

```
JNIEXPORT jfloatArray JNICALL Java_com_baidu_mdl_demo_MDL_
predictImage(JNIEnv *env, jclass thiz, jfloatArray buf)
```

这里需要传入图像的 3 维数组结构，真正执行预测的是 Net#predict(data) 方法，Net 模块是 MDL 网络管理模块，主要负责网络中各层 Layer 的初始化及管理工作。开发者在调用预测方法的时候，只需要调用对应 java 的 predictImage 方法，传入图像数据即可。

7. 预测完成，在 demo 的界面中返回预测耗时和结果。

性能和兼容性

框架	Caffe2	Tensorflow	ncnn	MDL(CPU)	MDL(GPU)
计算硬件	CPU	CPU	CPU	CPU	GPU
手机计算速度	慢	慢	很快	很快	极快
手机库大小	大	大	小	小	小
系统兼容性	Android&iOS	Android&iOS	Android&iOS	Android&iOS	iOS

总结

MDL 在 Android 和 iOS 系统上性能表现十分出色，并且 API 设计也很简

单易用，也支持其他的框架模型转换。总体来讲是一个非常优秀的移动端深度学习框架。

参考资料

- <https://github.com/baidu/mobile-deep-learning>

作者介绍

刘晓飞，TalkingData Android SDK 研发工程师，从事统计分析 Android SDK 需求研发，新功能调研，可视化埋点研究，负责 SDK 自动化构建打包工具的维护和升级。擅长领域为 Android 自动化构建工具应用和开发、Android 内存分析。

版权声明

InfoQ 中文站出品

AI in Mobile

©2018 北京极客邦科技有限公司

本书版权为北京极客邦科技有限公司所有，未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

出版：北京极客邦科技有限公司

北京市朝阳区来广营容和路叶青大厦北园 5 层

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系
editors@cn.infoq.com。

网 址：www.infoq.com.cn