Deep-learning with iOS and Android devices

Giulio Zaccaroni, Università di Bologna

Corso di Sistemi Digitali M
Stefano Mattoccia, Università di Bologna
Goal

- Executing TensorFlow deep neural networks on mobile devices
Why on-device (at the edge) prediction?

- Latency
- Privacy
- Server maintenance
- Server rental
- Offline-first experience
But there are also trade-offs you should keep in mind...

- **System utilization**: evaluating neural networks involves a lot of computation, which could increase battery power usage.
- **Application size**: models may take up multiple megabytes of space. If bundling large models in your APK would unduly impact your users, you may want to consider downloading the models after app installation.
iOS
Compatible iOS devices

- iPads and iPhones may have one, two or three cameras
- Accordingly, we can get mono, stereo or trinocular image streams
Both frameworks read neural networks only after conversion to specific formats (i.e., `tflite` and `mlmodel`).

- Allow to simplify deployment of deep learning networks on mobile devices.
- Although both aim to on-device training of neural networks, in CoreML it is available since version 3 while in TensorFlow Lite it is still under development.
- Both aim at supporting GPU hardware acceleration (enabling greater efficiency and speed) but TensorFlow Lite does not allow (with iOS) the use of the NPU (Neural Processing Unit).
- Both frameworks are constantly under development.
As of today, the best choice to achieve optimal performance with iOS devices is CoreML.

Then, firstly we need to convert TensorFlow model to CoreML.

Goal: conversion from tensorflow to a supported model.
Apple CoreML 1/2

- It was introduced in June during the WWDC 2017 conference
- It is a machine learning framework developed by Apple for MacOS, iOS, iPadOS, watchOS and tvOS to integrate machine learning in applications
- Core ML optimizes on-device performance by leveraging CPU, GPU, and Neural Engine minimizing memory footprint and power consumption
Apple CoreML 2/2

- It is the foundation for domain-specific frameworks and functionality
- It supports computer vision (images), natural language processing (text), speech (audio to text), and sound analysis (audio)
- Core ML itself builds on top of low-level primitives like Accelerate and BNNS, as well as Metal Performance Shaders
Basic Neural Network Subroutines (BNNS)

- It is a collection of functions that you use to implement and deep network
- Provides functions for creating, applying, and destroying three kinds of layers:
  - A **Convolution layer**, for each pixel in an input image, takes that pixel and its neighboring pixels and combines their values with weights from the training data to compute the corresponding pixel in the output image.
  - A **Pooling layer** produces a smaller output image from its input image by breaking the input image into smaller rectangular subimages; each pixel in the output is the maximum or average (your choice) of the pixels in the corresponding subimage. A pooling layer does not use training data.
  - A **fully connected layer** takes its input as a vector; this vector is multiplied by a matrix of weights from training data. The resulting vector is updated by the activation function.
Convolution layer

- Applies a convolution according to weights obtained in the training phase.
Pooling layer

- It reduces/decimates the image/feature size
- Pooling is agnostic to training
- For instance, the max-pooling layer acts as follows

\[
\begin{array}{cccc}
3 & 5 & 5 & 2 \\
1 & 8 & 4 & 2 \\
1 & 1 & 4 & 9 \\
3 & 2 & 6 & 9 \\
\end{array}
\rightarrow
\begin{array}{cc}
8 & 5 \\
3 & 9 \\
\end{array}
\]

max-pooling
Fully connected layer

- Multiplies an input feature vector by a matrix of weights obtained during the training phase.
Metal Performance Shaders (MPS)

- This framework contains a collection of highly optimized compute and graphics shaders that are designed to integrate easily and efficiently into your Metal app.
- These data-parallel primitives are specially tuned to take advantage of the unique hardware characteristics of each GPU family to ensure optimal performance.
MPS: supported functionalities

- The MPS framework supports the following functionalities:
  - Enables to apply high-performance filters and extract statistical and histogram data from images (and other data modality too)
  - Implements and run neural networks for machine learning training and inference
  - Solves systems of equations, factorize matrices and multiply matrices and vectors
  - Accelerate ray tracing with high-performance ray-geometry intersection testing
Hardware and software requirements

Neural network execution
- iOS > 11.0 (or iPadOS)

App development
- Xcode 11
- MacOS

Neural network conversion
- Python 2.7
- Linux or macOS
- Coremltools
- Tfcoreml
CoreML Model

Supported feature types:

- Image
- Int64
- Double
- Multidimensional array
- String
- Dictionary
- Sequence
You can:

- develop a CoreML model from the start with CreateML

or

- convert a model with coremltools from the following formats: Tensorflow, Caffe, Keras, libsvm, sklearn and xgboost
Converting a Tensorflow model to CoreML model: setup 1/2

1. Install pip
   
   ```
   $ easy_install pip
   ```

2. Install virtualenv
   
   ```
   $ sudo pip install virtualenv
   ```

3. Make a directory for the virtualenv
   
   ```
   $ mkdir mlvirtualenv
   $ cd mlvirtualenv
   ```

4. Create the virtualenv and activate it
   
   ```
   $ virtualenv pythonenv
   $ source pythonenv/bin/activate
   ```
5. Install coremltools (https://github.com/apple/coremltools)
   $ pip install -U coremltools
6. Install tfcoreml (https://github.com/tf-coreml/tf-coreml)
   $ pip install -U tfcoreml
Converting a Tensorflow model to CoreML model: usage


To convert a model from Tensorflow to CoreML just set

```python
configureFromConsole = True
```

in `converter.py` and launch it:

```
$ python converter.py
```

Case study: monocular depth estimation with PydNet

PydNet is a lightweight deep-network for depth perception from a single image

Source code: https://github.com/mattpoggi/pydnet
Source code (iOS and Android): https://github.com/FilippoAleotti/mobilePydnet
You need to:
- scale the image (or crop if needed)
- adapt pixel representation to neural network pixel representation
- transform the image to a pixel buffer
- execute code to predict result

```swift
let pydnet = Pydnet()
let input = PydnetInput(input: cvPixelBuffer)
let output = try pydnet.prediction(input: input)
```

let model = try VNCoreMLModel(for: Pydnet().model)

let request = VNCoreMLRequest(model: model, completionHandler: {
    [weak self] request, error in

    self?.processClassifications(for: request, error: error)
})

request.imageCropAndScaleOption = .centerCrop

Prediction from images 2/3

DispatchQueue.global(qos: .userInitiated).async {

    let handler = VNImageRequestHandler(ciImage: ciImage, orientation: orientation)

    do {
        try handler.perform([request])
    }
    catch {
        print("Failed to perform prediction. \n\n(error.localizedDescription)")
    }

}
guard let results = request.results else {
    print("Unable to perform prediction.\n\n(error!.localizedDescription)")
    return
}

let classifications = results as! [VNPixelBufferObservation]

for results in result {
    // Convert to UIImage and display
}
CoreML neural network quantization

- As models get more advanced, they can become large and take up significant storage space.
- Starting from iOS 12, weights can be encoded using any number of bits, all the way down to just 1-bit (binary weights).
- Thus, instead of having the continuous representation of values weights would have with floats, we actually end up with a discrete subset of the original representation.
- Of course, this will be a tradeoff between prediction accuracy and model size. There are multiple ways to choose the values representing those new, quantized weights (the simplest one consists of distribute them linearly).
Weights quantization
CoreML neural network quantization

The first image is the un-styled picture for reference. After it are displayed the outputs of a neural style transfer model for original, 16-bit, 8-bit, 6-bit, 4-bit, 3-bit, 2-bit and 1-bit quantized models.
Quantized models: results

Source: https://heartbeat.fritz.ai/reducing-coreml2-model-size-by-4x-with-quantization-in-ios12-b1c854651c4
The first image is the input image. The output of Pydnet with: 16-bit, 8-bit, 6-bit, 4-bit, 3-bit, 2-bit and 1-bit quantized models.
CoreML Pydnet neural network quantization: size
CoreML Pydnet neural network quantization: speed

Cast to float?

Unit: seconds
Quantization with coremltools

model = coremltools.models.MLModel('my_model.mlmodel')

quantized_model = quantization_utils.quantize_weights(model, 8, "linear")

Learn more at http://bit.ly/cmlquantutils
Server-side training

With this solution:

- You need to take data from user, set up some sort of cloud service, send that data to cloud service, and in the cloud create new model personalized on that data which will then deploy back to user.
- It creates the challenge of added expenses with making these cloud services, managing that service and creating an infrastructure that could scale to million of users.
- Obviously, users have to expose their data to others.
On-device training

A much better solution consists of simply taking a general model and, with the data created by the user, personalize the original model

- The data stays private, it never leaves the device
- There's no need for a server to do this kind of interactivity
- On the other hand, updating a model is not straightforward and requires appropriate methodologies to (self) update the network
Marking a layer as *updatable*

```python
import coremltools

coreml_model_path = 'model.mlmodel'
spec = coremltools.utils.load_spec(coreml_model_path)
builder = coremltools.models.neural_network.NeuralNetworkBuilder(spec=spec)

model_spec = builder.spec
builder.make_updatable(['layer1', 'layer2'])

mlmodel_updatable = MLModel(model_spec)
mlmodel_updatable.save(mlmodel_updatable_path)
```
var batchInputs: [MLFeatureProvider] = []

let featureProvider = PydnetTrainingInput(image: pixelBuffer,
                                          depth: depthPixelBuffer)

batchInputs.append(featureProvider)

let trainingData = MLArrayBatchProvider(array: batchInputs)
let updateTask = try MLUpdateTask(forModelAt: updatableModelURL,
  trainingData: trainingData,
  configuration: model.configuration,
  completionHandler: { context in
  // Training has finished, save the updated model
 })
updateTask.resume()
The application is available for download: installing AppML

You can try your CoreML models (both stereo and mono supported) on AppML

Download TestFlight
https://apple.co/2DGr6q9

Download AppML
https://bitly.com/appml_beta
Adding a neural network to AppML

1. Open neural network file
2. Tap share icon
3. Search for “Copy to AppML” and press it
4. You’re ready to go!
Summary

● CoreML is a powerful tool for optimizing ML networks for iOS devices
● Achieves outstanding performance leveraging Apple architectures
● Enables seamless integration of pre-trained CNNs in iOS devices
● On-line training paves the way for new exciting developments
Android
Compatible Android devices

- Every android device with one, two or three cameras
TensorFlow Lite consists of two main components:

- The TensorFlow Lite interpreter, which runs specially optimized models on many different hardware types.
- The TensorFlow Lite converter, which converts TensorFlow models into an efficient form for use by the interpreter, and can introduce optimizations to improve binary size and performance.
Android Neural Networks API

Android’s neural network runtime can distribute the computation workload across available on-device processors, including:

- dedicated neural network hardware
- graphics processing units (GPUs)
- digital signal processors (DSPs)

For Android devices that lack a specialized vendor driver, the NNAPI runtime executes the requests on the CPU.

A **DSP** is a microprocessor chip with an architecture optimized for digital signal processing.
Hardware and software requirements

Neural network execution
- Android > 5.0

App development
- Android Studio 3.5

Neural network conversion
- Tensorflow
- Python > 2.7
The set of TensorFlow Lite operations is smaller than TensorFlow's, furthermore many ops do not yet support types like tf.float16 and strings.

By the way float32, uint8 and int8 are fully supported.
import tensorflow as tf
localpb = 'optimized_pydnet++.pb'
tflite_file = 'optimized_pydnet++.tflite'
converter = tf.compat.v1.lite.TFLiteConverter.from_frozen_graph(
    localpb,
    ['im0'],
    ['PSD/resize_images/ResizeBilinear'],
    input_shapes={'im0' : [1, 448, 640, 3]}
)
tflite_model = converter.convert()
open(tflite_file, 'wb').write(tflite_model)
Converting a TensorFlow model to TensorFlow Lite: optimizations

16-bit Quantization (2x smaller, potential gpu acceleration)

You can reduce the size of a floating point model by quantizing the weights to float16, the IEEE standard for 16-bit floating point numbers. The advantages are:

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.lite.constants.FLOAT16]
```

!!! A float16 quantized model will "dequantize" the weights values to float32 when run on the CPU. The GPU delegate will not perform this dequantization, since it can operate on float16 data.

Converting a TensorFlow model to TensorFlow Lite: optimizations

8-bit Quantization (4x smaller, 2-3x speedup, no gpu support)

The simplest form of post-training quantization quantizes only the weights from floating point to 8-bits of precision (also called "hybrid" quantization), at inference, weights are converted from 8-bits of precision to floating point and computed using floating-point kernels. This conversion is done once and cached to reduce latency.

```python
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
```

The model still uses float input and output for convenience, you can enforce integer input and output by adding the following lines before you convert:

```python
compiler.inference_input_type = tf.uint8
compiler.inference_output_type = tf.uint8
```

Colab notebook  
The first image is the input image. The output of Pydnet with 16-bit and 8-bit quantized models.
TensorFlow Lite Pydnet neural network quantization: size
TensorFlow Lite Pydnet neural network quantization: speed

Unit: seconds
CoreML and TensorFlow Lite comparison: Pydnet

Test run on a MacBook Pro 15" (CPU: Core i7 2.6 GHz Coffee Lake 45W, 16 GB RAM, GPU: Radeon Pro 560X 4 GB)
Prediction with TensorFlow Lite

1. Add “tflite” or “lite” file to “assets” folder
2. Add the following code to app/build.gradle to avoid compression:

   ```java
   aaptOptions {
     noCompress "tflite"
     noCompress "lite"
   }
   ```

3. Initialize interpreter:

   ```java
   Interpreter.Options tfliteOptions = new Interpreter.Options();
   Interpreter tfLite = new Interpreter(loadModelFile(context.getAssets(), "optimized_pydnet++.tflite"),
   tfliteOptions);
   ```

4. Run interpreter:

   ```java
   outputByteBuffer = ByteBuffer.allocateDirect(resolution.getHeight() * resolution.getWidth() * 4);
   tfLite.run(inputByteBuffer, outputByteBuffer);
   ```
Prediction with TensorFlow Lite: Interpreter Options

Android NNAPI

```java
val interpreterOptions = Interpreter.Options()
interpreterOptions.setUseNNAPI(true)
```

Number of threads

```java
val interpreterOptions = Interpreter.Options()
interpreterOptions.setNumThreads(4)
```
Prediction with TensorFlow Lite: Interpreter Options

GPU Delegate

- Add tensorflow-lite-gpu package in the dependencies block (app/build.gradle)
- Add delegate to interpreter options:

```java
Interpreter.Options interpreterOptions = new Interpreter.Options()
GpuDelegate delegate = new GpuDelegate()
interpreterOptions.addDelegate(delegate)
```
References

- Apple, CoreML Documentation, https://apple.github.io/coremltools/coremlspecification/
- Neural Networks API | Android NDK, https://developer.android.com/ndk/guides/neuralnetworks
- M. Poggi, F. Aleotti, F. Tosi, S. Mattoccia, “Towards real-time unsupervised monocular depth estimation on CPU”, IROS 2018
- F. Aleotti, “Introduction to Tensorflow”, available [here](#)