An On-device Deep Neural Network for Face Detection

Vol. 1, Issue 7 • November 2017

by Computer Vision Machine Learning Team

Apple started using deep learning for face detection in iOS 10. With the release of the Vision framework, developers can now use this technology and many other computer vision algorithms in their apps. We faced significant challenges in developing the framework so that we could preserve user privacy and run efficiently on-device. This article discusses these challenges and describes the face detection algorithm.

Introduction

Apple first released face detection in a public API in the Core Image framework through the CIDetector class. This API was also used internally by Apple apps, such as Photos. The earliest release of CIDetector used a method based on the Viola-Jones detection algorithm [1]. We based subsequent improvements to CIDetector on advances in traditional computer vision.

With the advent of deep learning, and its application to computer vision problems, the state-of-the-art in face detection accuracy took an enormous leap forward. We had to completely rethink our approach so that we could take advantage of this paradigm shift. Compared to traditional computer vision, the learned models in deep learning require orders of magnitude more memory, much more disk storage, and more computational resources.

As capable as today's mobile phones are, the typical high-end mobile phone was not a viable platform for deep-learning vision models. Most of the industry got around this problem by providing deep-learning solutions through a cloud-based API. In a cloud-based solution, images are sent to a server for analysis using deep learning inference to detect faces. Cloud-based services typically use powerful desktop-class GPUs with large amounts of memory available. Very large network models, and potentially ensembles of large models, can run on the server side, allowing clients (which could be mobile phones) to take advantage of large deep learning architectures that would be impractical to run locally.

Apple's iCloud Photo Library is a cloud-based solution for photo and video storage. However, due to Apple's strong commitment to user privacy, we couldn't use iCloud servers for computer vision computations. Every photo and video sent to iCloud Photo Library is encrypted on the device before it is sent to cloud storage, and can only be decrypted by devices that are registered with the iCloud account. Therefore, to bring deep learning based computer vision solutions to our customers, we had to address directly the challenges of getting deep learning algorithms running on iPhone.

We faced several challenges. The deep-learning models need to be shipped as part of the operating system, taking up valuable NAND storage space. They also need to be loaded into RAM and require significant computational time on the GPU and/or CPU. Unlike cloud-based services, whose resources can be dedicated solely to a vision problem, on-device computation must take place while sharing these system resources with other running applications. Finally, the computation must be efficient enough to process a large Photos library in a reasonably short amount of time, but without significant power usage or thermal increase.

The rest of this article discusses our algorithmic approach to deep-learning-based face detection, and how we successfully met the challenges to achieve state-of-the-art accuracy. We discuss:

- how we fully leverage our GPU and CPU (using BNNS and Metal)
- memory optimizations for network inference, and image loading and caching
- how we implemented the network in a way that did not interfere with the multitude of other simultaneous tasks expected of iPhone.

Moving From Viola-Jones to Deep Learning

In 2014, when we began working on a deep learning approach to detecting faces in images, deep convolutional networks (DCN) were just beginning to yield promising results on object detection tasks. Most prominent among these was an approach called “OverFeat” [2] which popularized some simple ideas that showed DCNs to be quite efficient at scanning an image for an object.

OverFeat drew the equivalence between fully connected layers of a neural network and convolutional layers with valid convolutions of filters of the same spatial dimensions as the input. This work made clear that a binary classification network of a fixed receptive field (for example 32x32, with a natural stride of 16 pixels) could be efficiently applied to an arbitrary sized image (for example, 320x320) to produce an appropriately sized output map (20x20 in this example). The OverFeat paper also provided clever recipes to produce denser output maps by effectively reducing the network stride.

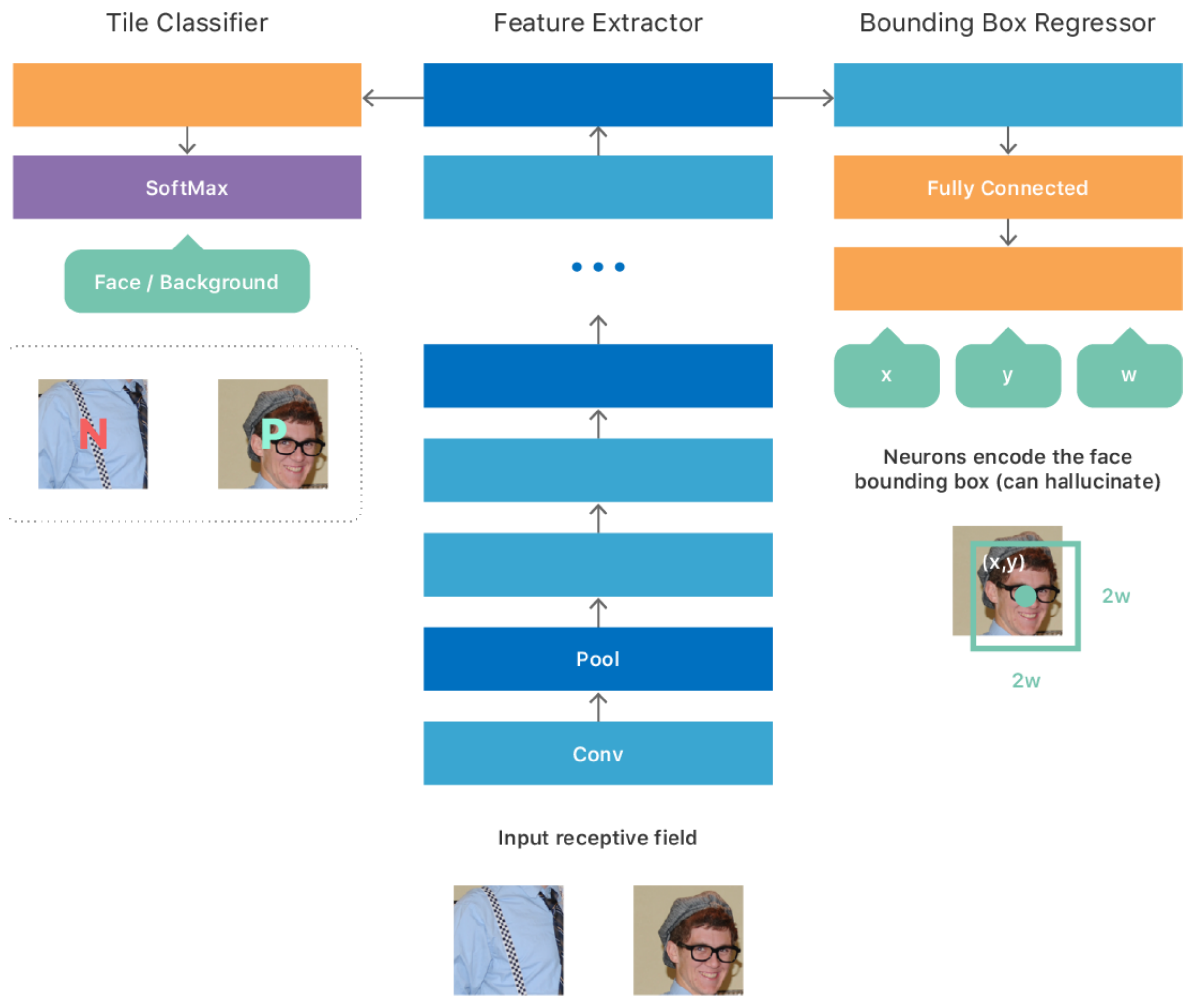
We built our initial architecture based on some of the insights from the OverFeat paper, resulting in a fully convolutional network (see Figure 1) with a multitask objective comprising of:

- a binary classification to predict the presence or absence of a face in the input, and
- a regression to predict the bounding box parameters that best localized the face in the input.

We experimented with several ways of training such a network. For example, a simple procedure for training is to create a large dataset of image tiles of a fixed size corresponding to the smallest valid input to the network such that each tile produces a single output from the network. The training dataset is ideally balanced, so that half of the tiles contain a face (positive class) and the other half do not contain a face (negative class). For each positive tile, we provide the true location (x, y, w, h) of the face. We train the network to optimize the multitask objective described previously.

Once trained, the network is able to predict whether a tile contains a face, and if so, it also provides the coordinates and scale of the face in the tile.

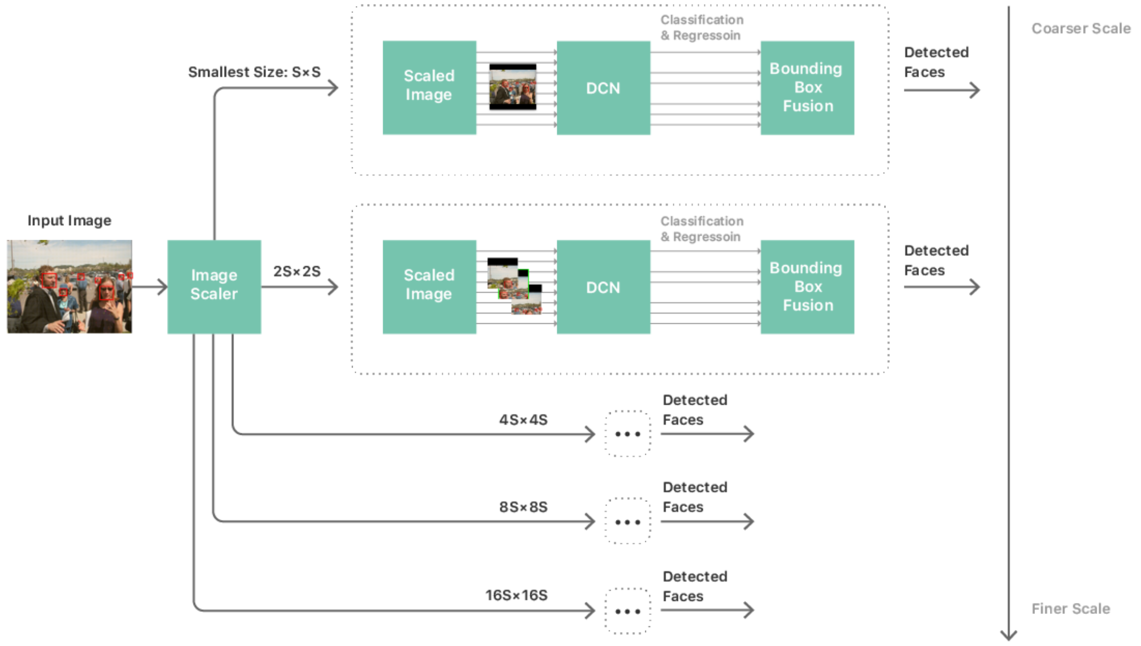
Figure 1. A revised DCN architecture for face detection



Since the network is fully convolutional, it can efficiently process an arbitrary sized image and produce a 2D output map. Each point on the map corresponds to a tile in the input image and contains the prediction from the network regarding the presence of or absence of a face in that title and its location/scale within the input tile (see inputs and outputs of DCN in Figure 1).

Given such a network, we could then build a fairly standard processing pipeline to perform face detection, consisting of a multi-scale image pyramid, the face detector network, and a post-processing module. We needed a multi-scale pyramid to handle faces across a wide range of sizes. We apply the network to each level of the pyramid and candidate detections are collected from each layer. (See Figure 2.) The post processing module then combines these candidate detections across scales to produce a list of bounding boxes that correspond to the network's final prediction of the faces in the image.

Figure 2. Face detection workflow.



This strategy brought us closer to running a deep convolutional network on device to exhaustively scan an image. But network complexity and size remained key bottlenecks to performance. Overcoming this challenge meant not only limiting the network to a simple topology, but also restricting the number of layers of the network, the number of channels per layer, and the kernel size of the convolutional filters. These restrictions raised a crucial problem: our networks that were producing acceptable accuracy were anything but simple, most going over 20 layers and consisting of several network-in-network [3] modules. Using such networks in the image scanning framework described previously would be completely infeasible. They led to unacceptable performance and power usage. In fact, we would not even be able to load the network into memory. The challenge then was how to train a simple and compact network that could mimic the behavior of the accurate but highly complex networks.

We decided to leverage an approach, informally called “teacher-student” training[4]. This approach provided us a mechanism to train a second thin-and-deep network (the “student”), in such a way that it matched very closely the outputs of the big complex network (the “teacher”) that we had trained as described previously. The student network was composed of a simple repeating structure of 3x3 convolutions and pooling layers and its architecture was heavily tailored to best leverage our neural network inference engine. (See Figure 1.)

Now, finally, we had an algorithm for a deep neural network for face detection that was feasible for on-device execution. We iterated through several rounds of training to obtain a network model that was accurate enough to enable the desired applications. While this network was accurate and feasible, a tremendous amount of work still remained to make it practical for deploying on millions of user devices.

Optimizing the Image Pipeline

Practical considerations around deep learning factored heavily into our design choices for an easy-to-use framework for developers, which we call Vision. It became quickly apparent that great algorithms are not enough for creating a great framework. We had to have a highly optimized imaging pipeline.

We did not want developers to think about scaling, color conversions, or image sources. Face detection should work well whether used in live camera capture streams, video processing, or processing of images from disc or the web. It should work regardless of image representation and format.

We were concerned with power consumption and memory usage, especially for streaming and image capture. We worried about memory footprint, such as the large one needed for a 64 Megapixel panorama. We addressed these concerns by using techniques of partial subsampled decoding and automatic tiling to perform computer vision tasks on large images even with non-typical aspect ratios.

Another challenge was colorspace matching. Apple has a broad set of colorspace APIs but we did not want to burden developers with the task of color matching. The Vision framework handles color matching, thus lowering the threshold for a successful adoption of computer vision into any app.

Vision also optimizes by efficient handling and reuse of intermediates. Face detection, face landmark detection, and a few other computer vision tasks work from the same scaled intermediate image. By abstracting the interface to the algorithms and finding a place of ownership for the image or buffer to be processed, Vision can create and cache intermediate images to improve performance for multiple computer vision tasks without the need for the developer to do any work.

The flip side was also true. From the central interface perspective, we could drive the algorithm development into directions that allow for better reusing or sharing of intermediates. Vision hosts several different, and independent, computer vision algorithms. For the various algorithms to work well together, implementations use input resolutions and color spaces that are shared across as many algorithms as possible

Optimizing for On-device Performance

The joy of ease-of-use would quickly dissipate if our face detection API were not able to be used both in real time apps and in background system processes. Users want face detection to run smoothly when processing their photo libraries for face recognition, or analyzing a picture immediately after a shot. They don't want the battery to drain or the performance of the system to slow to a crawl. Apple's mobile devices are multitasking devices. Background computer vision processing therefore shouldn't significantly impact the rest of the system's features.

We implement several strategies to minimize memory footprint and GPU usage. To reduce memory footprint, we allocate the intermediate layers of our neural networks by analyzing the compute graph. This allows us to alias multiple layers to the same buffer. While being fully deterministic, this technique reduces memory footprint without impacting the performance or allocations fragmentation, and can be used on either the CPU or GPU.

For Vision, the detector runs 5 networks (one for each image pyramid scale as shown in Figure 2). These 5 networks share the same weights and parameters, but have different shapes for their input, output, and intermediate layers. To reduce footprint even further, we run the liveness-based memory optimization algorithm on the joint graph composed by those 5 networks, significantly reducing the footprint. Also, the multiple networks reuse the same weight and parameter buffers, thus reducing memory needs.

To achieve better performance, we exploit the fully convolutional nature of the network: All the scales are dynamically resized to match the resolution of the input image. Compared to fitting the image in square network retinas (padded by void bands), fitting the network to the size of the image allows us to reduce drastically the number of total operations. Because the topology of the operation is not changed by the reshape and the high performance of the rest of the allocator, dynamic reshaping does not introduce performance overhead related to allocation.

To ensure UI responsiveness and fluidity while deep neural networks run in background, we split GPU work items for each layer of the network until each individual time is less than a millisecond. This allows the driver to switch contexts to higher priority tasks in a timely manner, such as UI animations, thus reducing and sometimes eliminating frame drop.

Combined, all these strategies ensure that our users can enjoy local, low-latency, private deep learning inference without being aware that their phone is running neural networks at several hundreds of gigaflops per second.

Using Vision Framework

Did we accomplish what we set as our goal of developing a performant, easy-to-use, face detection API? You can try the Vision framework and judge for yourself. Here's how to get started:

- Watch the WWDC presentation: Vision Framework: Building on Core ML.
- Read the Vision Framework Reference.
- Try out the Core ML and Vision: Machine Learning in iOS 11 Tutorial. [5]

References

[1] Viola, P. and Jones, M.J. **Robust Real-time Object Detection Using a Boosted Cascade of Simple Features**. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, 2001.

[2] Sermanet, Pierre, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann LeCun. **OverFeat: Integrated Recognition, Localization and Detection Using Convolutional Networks**. *arXiv:1312.6229* [Cs], December, 2013.

[3] Lin, Min, Qiang Chen, and Shuicheng Yan. **Network In Network**. *arXiv:1312.4400* [Cs], December, 2013.

[4] Romero, Adriana, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. **FitNets: Hints for Thin Deep Nets**. *arXiv:1412.6550* [Cs], December, 2014.

[5] Tam, A. **Core ML and Vision: Machine learning in iOS Tutorial**. Retrieved from https://www.raywenderlich.com, September, 2017.