
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Table of Contents: Extended

Prerequisites

Some tutorials and workshops at RWDevCon 2018 have prerequisites.

If you plan on attending any of these tutorials or workshops, **be sure to complete the steps below before attending the tutorial**.

If you don't, you might not be able to follow along with those tutorials.

> **Note**: All talks require **Xcode 9.2** installed, unless specified otherwise (such as in the ARKit tutorial and workshop).

W2: Machine Learning Workshop

First, make sure you have at least 3 GB of free disk space.

Docker & openface Image

Download **Docker Community Edition for Mac** from:

- https://store.docker.com/editions/community/docker-ce-desktop-mac

Install and start **Docker**, then open **Terminal**, and enter the following command:

```
docker pull bamos/openface
```

When this command finishes, enter the following command:

```
docker images
```

You should see `bamos/openface` listed under `REPOSITORY`. Its `SIZE` is 2.54 GB.

openface/test-images

In **Finder**, create a folder named **openface**, somewhere convenient, then create a subfolder named **test-images**. Copy the images from **Demo2/images** into **test-images**. Add images of yourself and people you know, in subfolders named, for example, *ray-wenderlich*.

You need just a couple of images for each person.

Make sure only one face appears in each image. You don't need to crop the image around the face, and the face doesn't need to be looking straight at the camera.

openface/training-images

Create another subfolder of **openface**, named **training-images**, then create named subfolders in **training-images** — these folders should match *your additional* folders in **openface/test-images**, although you can have test image folders that don't have a matching training image folder.

You need about **20 images** for each person.

Make sure only one face appears in each image. You don't need to crop the image around the face, and the face doesn't need to be looking straight at the camera.

Lightning Cable

One of the sample apps only runs on an iOS device, so bring a lightning cable — sometimes Xcode's wireless debugging doesn't work until you've first run the app via a cable.

W2: Machine Learning Workshop

With Apple's new CoreML and Vision frameworks, you can now add machine learning AI to your apps. In this hands-on workshop, you'll learn what machine learning actually is, how to train a model, and integrate it into an app.

62 Machine Learning Workshop: Demo 1

By Audrey Tam

In this demo, you will use the Vision framework to find faces in an image and in a video stream.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the **FaceFinder** starter project in **Demo1/starter/part1**.

1) Add Images

Build and run the **FaceFinder** starter project. then switch to the **Photos** app, and drag photos with people into it. Use your own photos, or use the images in the starter project's **images** folder.

Switch back to the app, tap the **+** button, and select an image. Tap its thumbnail in the collection view, to open it in the image view controller.

So far, nothing more happens. Stop the app.

2) Create & Run Face Request

Open **ImageViewController.swift**, and replace `viewWillAppear(_:)` with:

```
override func viewWillAppear(_ animated: Bool) {
  super.viewWillAppear(animated)
  removeMask()
  imageView.image = image
  guard let cgImage = image.cgImage else {
    fatalError("Can't create CGImage.")
```

```
    }

    // Step 2
   let faceRequest = VNDetectFaceRectanglesRequest(
      completionHandler: handleFaces
    )
    let orientation = CGImagePropertyOrientation(image.imageOrientation)
    let imageHandler = VNImageRequestHandler(cgImage: cgImage,
      orientation: orientation)
    DispatchQueue.global(qos: .userInteractive).async {
      do {
        try imageHandler.perform([faceRequest])
      } catch {
        print("Image handler error \(error).")
      }
    }
  }
```

3) Define Face Request Completion Handler

Replace `handleFaces(request:error:)` with:

```
  func handleFaces(request: VNRequest, error: Error?) {
    guard let observations = request.results as? [VNFaceObservation] else {
        fatalError("Unexpected result type from request.")
    }

    DispatchQueue.main.async {
      self.drawFaceBoxes(observations: observations)
    }
  }
```

4) Add a Breakpoint

In `handleFaces(request:error:)`, add a breakpoint, then edit the breakpoint to log the method name, and automatically continue:

Do the same in `drawFaceBoxes(observations:)`, in the `for face in observations` loop.

Build and run, then select an image from the collection view. Wait a while to see the breakpoint log messages and the green boxes appear.

5) Track Faces in Video Stream

Now open the **part2** starter project **FaceTracker**.

a) In the extension, add the following to the end of the delegate method

`captureOutput(_:didOutput:from:)`:

```
let faceRequest = VNDetectFaceRectanglesRequest(completionHandler:
handleFaces)
let imageHandler = VNImageRequestHandler(cvPixelBuffer: pixelBuffer,
  orientation: exifOrientation, options: requestOptions)
DispatchQueue.global(qos: .userInteractive).async {
  do {
    try imageHandler.perform([faceRequest])
  } catch {
    print("Image handler error \(error).")
  }
}
```

b) In **VideoViewController.swift**, replace the `handleFaces(request:error:)` stub with:

```
func handleFaces(request: VNRequest, error: Error?) {
  guard let observations = request.results as? [VNFaceObservation] else {
    fatalError("Unexpected result type from request.")
  }
  if let newObservation = observations.first {
    DispatchQueue.main.async {
      self.removeMask()
      self.drawHighlight(boundingBox: newObservation.boundingBox)
    }
  }
}
```

6) Build & Run on iPhone

Select the project in the project navigator. In the **Identity** section, change the **Bundle Identifier** to something unique to you. In the **Signing** section, select a **Team** — it can be your Apple ID; it doesn't have to be a paid developer account.

Build and run the app on your iOS device: point the back camera at someone nearby, and wait a little while for a green box to appear around their face. Then move your iPhone around, or ask your friend to move around, and watch the green box follow their face.

> **Note:** You need a reasonable amount of light, and a reasonably new iPhone.

7) That's it!

Congrats, at this time you should have a good understanding of using Vision to find and track faces! It's time to move on to using Core ML.

Resources

- An On-device Deep Neural Network for Face Detection(apple.co/2j11AkR): Apple's Machine Learning Journal post about their face detection algorithm, with my raywenderlich.com tutorial Core ML and Vision: Machine Learning in iOS 11 Tutorial(bit.ly/2HT2zPX) in the References!
- Jeffrey Bergier's ObjectTracker project(bit.ly/2qTb6eC): An object tracking app that uses `VNSequenceRequestHandler`.
- iOS 11 By Tutorials(bit.ly/2wcpx07): Demo 1 FaceFinder app is adapted from the sample app in Chapter 9: CoreML & Vision.
- Wei Chieh Tseng's AppleFaceDetection project(bit.ly/2qUNTba): Demo 1 FaceTracker app uses code from this fully featured face detection app.

Machine Learning Workshop: Demo 2

By Audrey Tam

In this demo, you will add CoreML models to the project, to recognize aligned faces.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo2\starter**. It's similar to **FaceFinder**, but I've added a label to its `ImageViewController` scene to display the prediction. I've also stripped out all the face detection and box drawing code.

1) Preprocess Images

The **Prerequisites** document gave you instructions to:

- download and install **Docker** and the `bamos/openface` image;
- create an **openface** folder with subfolders named **test-images** and **training-images**;
- copy the images from **Demo2/images** into **test-images**, and add images of yourself and other people, in named subfolders;
- add named subfolders to **openface/training-images** for yourself and (some of) the people you added, each with ~20 images — you'll use the training images in Demo 3.

At the beginning of this workshop, you started **Docker**, and checked that you have the **openface** image. Now run this command in **Terminal** (it's a long command — enter it as a **single** line):

```
docker run --rm -it -v /Users:/host/Users bamos/openface /bin/bash
```

At the Docker root prompt, enter these commands:

```
export OFPATH="/host<path to your openface folder>/"
```

```
cd /root/openface/
```

This one's a long command — be sure to enter it as a **single** line:

```
./util/align-dlib.py $OFPATH/test-images/ align outerEyesAndNose $OFPATH/
aligned-test-images/ --size 96
```

This step converts these test images:

into these aligned test images:

Something similar has happened to your additional images.

When the test images are ready, load them into the simulator device's Photos app, so they're there when you test the finished app.

2) Add Models

Drag **OpenFace.mlmodel** and **OpenFaceClassifier.mlmodel** from the **Demo2/models** folder into the project navigator.

Select each model, then click its arrow under **Model Class** to see the generated Swift class. If it says *Build target "FaceIDer" to generate Swift model class*, press **Command-B** to do that.

> **Note:** Sometimes Xcode says `OpenFace` and/or `OpenFaceClassifier` are unresolved identifiers, but you can still build and run the app, and it works just fine.

3) Add Faces

In **ImageViewController.swift**, below the outlets, add this array of known faces (the ones I trained OpenFaceClassifier.mlmodel to recognize):

```
let faces = ["Emma Thompson", "Jackie Chan", "Lisa Jackson",
  "Michelle Yeoh", "Pallavi Sharda", "Steve Jobs", "Tim Cook"]
```

In the simulator, open **Photos**, and add images from your **openface/aligned-test-images** folder.

4) Create OpenFace Request

a) In **ImageViewController.swift**, add this lazy var:

```
lazy var openFaceRequest: VNCoreMLRequest = {
  do {
    let openFaceModel = try VNCoreMLModel(for: OpenFace().model)
    return VNCoreMLRequest(model: openFaceModel,
      completionHandler: self.handleOpenface)
  } catch {
    fatalError("Can't load Vision ML model: \(error).")
  }
}()
```

b) Next, fill in the request's completion handler stub:

```
func handleOpenface(request: VNRequest, error: Error?) {
  guard let results = request.results
    as? [VNCoreMLFeatureValueObservation],
    let topResult = results.first?.featureValue.multiArrayValue else
{  // embedding
      fatalError("Unexpected result type from VNCoreMLRequest.")
  }
  classifyFace(topResult)
}
```

5) Define classifyFace(_:)

Instantiate the classifier model, and fill in the `classifyFace(_:)` stub:

```
let classifierModel = OpenFaceClassifier()
func classifyFace(_ embedding: MLMultiArray) {
  // Call classifier model's prediction method
  do {
    let classification = try classifierModel.prediction(input: embedding)
    let index = classification.classLabel
    let probability = Int(100 * classification.classProbability[index]!)
    DispatchQueue.main.async {
      self.predictionLabel.text = "\(self.faces[Int(index)]) with \
(probability)% confidence"
    }
  } catch {
    fatalError("Classifier can't make prediction: \(error).")
  }
}
```

6) Define predictIdentity(image:)

a) Next, fill in the `predictIdentity(image:)` stub, to run `openFaceRequest`:

```
func predictIdentity(image: UIImage) {
  guard let ciImage = CIImage(image: image) else {
    fatalError("Can't create CIImage from UIImage.")
  }
  let imageHandler = VNImageRequestHandler(ciImage: ciImage)
  DispatchQueue.global(qos: .userInteractive).async {
    do {
      try imageHandler.perform([self.openFaceRequest])
    } catch {
      fatalError("Can't perform image request: \(error).")
    }
  }
}
```

b) Finally, in `viewWillAppear(_:)`, after the call to `super`, add the following:

```
predictIdentity(image: image)
```

Build and run, then add an aligned test image to the collection view, and select it. You should now see a prediction label at the top when selecting an image. For images of your additional people, the model will find the closest match to a known face.

7) That's it!

Congrats, at this time you should have a good understanding of using CoreML with Vision!

Take a break, and then it's time to learn how to train your model.

> **Note**: I've included `predictIdentityWithoutVision()`, to show how you could use the OpenFace model without Vision. It uses a `UIImage` extension to create a `CVPixelBuffer` object, passes this to the OpenFace model's prediction method, then passes the appropriate output to the classifier model. Uncomment the code, then uncomment the call to `predictIdentityWithoutVision()`, and comment out the call to `predictIdentity(image:)`, to see how it performs: it still gets the correct identities, but most of the confidence values are lower.

Resources

- Beginning Machine Learning with Keras & Core ML(bit.ly/2vAilg6): ML in a nutshell + how to train a convolutional neural network, and convert it to CoreML.
- Lumina on GitHub(bit.ly/2qWtC4X): A camera designed in Swift for easily integrating CoreML models.

- Apple's IBM Watson Services for Core ML announcement(apple.co/2FbSL0v): Build apps that access powerful Watson capabilities.
- OpenFace repo(bit.ly/1niqDjY): A Python and Torch implementation of face recognition with deep neural networks.

64 Machine Learning Workshop: Demo 3

By Patrick Kwete

In this demo, you will create a new CoreML face recognition classification model for your app.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

If you're continuing from **Demo2**, you've already started Docker and run these commands in Terminal:

```
docker pull bamos/openface
```

```
docker run --rm -it -v /Users:/host/Users bamos/openface /bin/bash
```

And at the Docker root prompt:

```
export OFPATH="/host/Users/<path to your openface folder>/"
```

```
cd /root/openface
```

1) Preprocess & process your images.

Run the openface scripts from inside the openface root directory:

a.) Align your images:

```
./util/align-dlib.py $OFPATH/training-images/ align outerEyesAndNose
$OFPATH/aligned-images/ --size 96
```

This will create a new ./aligned-images/ subfolder with a cropped and aligned version of each of your training images.

b.) Generate the representations from the aligned images:

```
./batch-represent/main.lua -outDir $OFPATH/generated-embeddings/ -data
$OFPATH/aligned-images/
```

The ./generated-embeddings/ sub-folder will contain csv files with the labels and embeddings for each image.

2) Create your functional face recognizer.

a.) Install CoreMLtools

```
pip install coremltools
```

b) Add the following lines to `./demos/classifier.py` :

i.) Include the Coremltools import statement :

```
import coremltools
```

ii.) Add coreml conversion lines at the end of the 'train' function. The 'train' function is the code block below `def train(args):` :

```
 coreml_model = coremltools.converters.sklearn.convert(clf)
 // Note: replace $OFPATH with your actual openface folder path
 coreml_model.save('$OFPATH/generated-embeddings/
OpenFaceClassifier.mlmodel')
```

Remember to replace `$OFPATH` with your actual openface folder path.

c.) Train & convert your face detection model:

```
python ./demos/classifier.py train $OFPATH/generated-embeddings/
```

This will generate a new file called `$OFPATH/generated-embeddings/OpenFaceClassifier.mlmodel` . This has the model converted to CoreML format to recognize new faces in the app.

3) Replace the CoreML classifier model in FaceIDer

Open the **FaceIDer** project in **Demo2/final**.

Replace the previous **OpenFaceClassifier.mlmodel** in the project navigator with your new **OpenFaceClassifier.mlmodel**.

Select the model, then click its arrow under **Model Class** to see the generated Swift class. If it says *Build target "FaceIDer" to generate Swift model class*, press

Command-B to do that.

> **Note:** Sometimes Xcode says `OpenFace` and/or `OpenFaceClassifier` are unresolved identifiers, but you can still build and run the app, and it works just fine.

4) Modify faces array

Open **ImageViewController.swift**, and locate the `faces` array, under the **Step 3** comment.

Open **generated-embeddings/labels.csv** and modify the `faces` array to match: put the names into the array in the same order they appear. Note that the labels numbering starts at 1.

Build and run, then add one of your aligned test images to the collection view, and select it. You should now see a personalized prediction label. For test images of anyone you didn't include when you trained the model, the model will find the closest match to a known face.

5) That's it!

Congrats, at this time you should have a good understanding of creating a new CoreML face recognition classification model for your app!