Sample Code

Training a Create ML Model to Classify Flowers

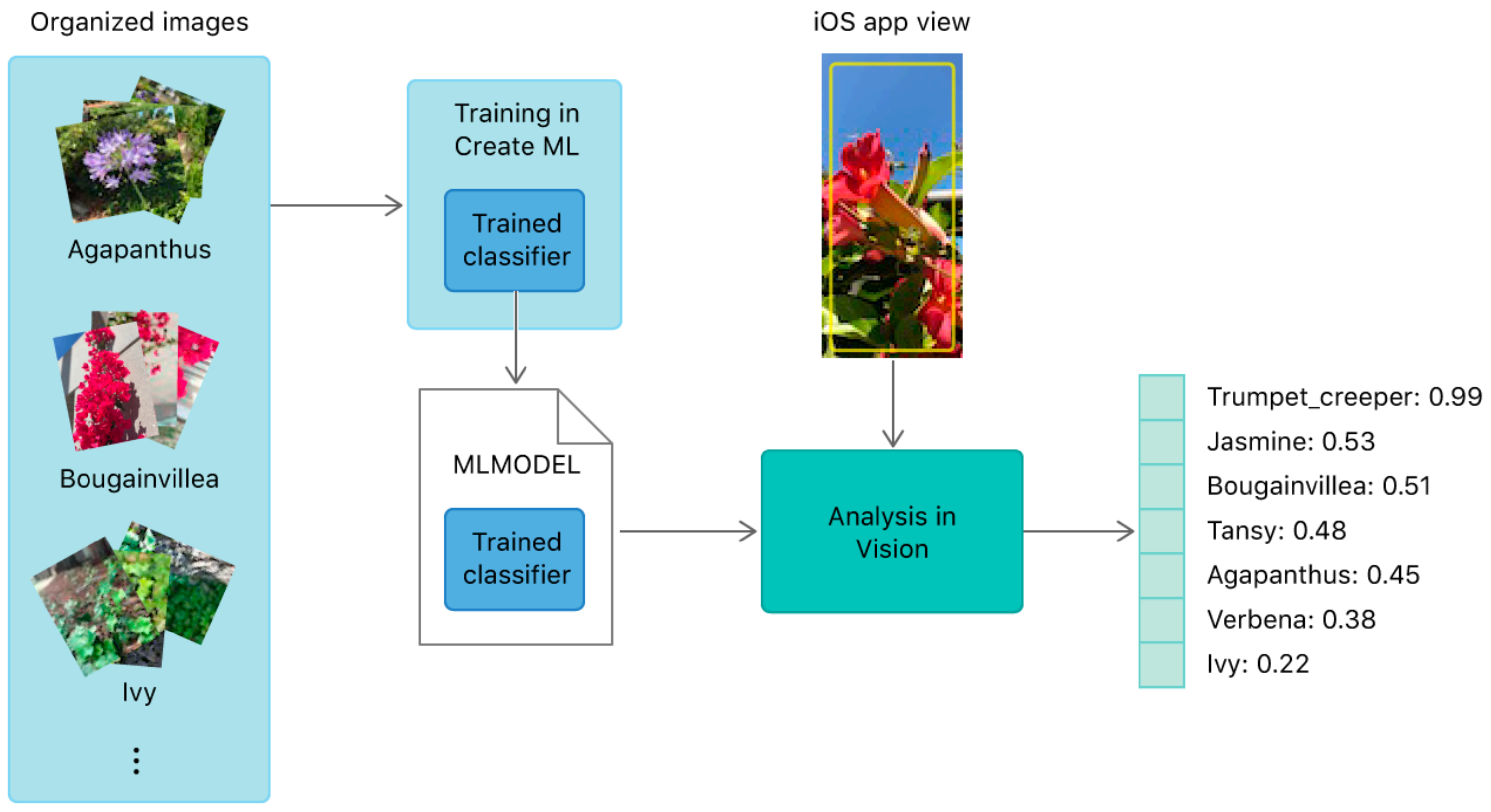
Train a flower classifier using Create ML in Swift Playgrounds, and apply the resulting model to real-time image classification using Vision.

Download

iOS 12.0+ | iPadOS 12.0+ | Xcode 11.3+

Overview

To classify images in real time, you need a classification model with the categories you'd like identified, and a way to capture images to feed to the classifier.



This sample code project contains two components: a Create ML model you train in Swift Playgrounds, and the iOS app, FlowerShop, which you use to classify different flower types. This project uses the same code as the robot shop demo in the WWDC 2018 session Vision with Core ML.

Getting Started

To see Create ML in action, run the Swift Playground `Training/ImageClassifier Playground.playground` in Xcode 10 on a Mac running macOS 10.14 or later. This Swift Playground performs classification on the training set and generates the Create ML model `Image Classifier.mlmodel`.

The sample app, FlowerShop, requires the following:

- Xcode 10.
- An iOS device running iOS 12 or later.

Because Xcode can't access the camera, FlowerShop won't work in Simulator.

Train a Custom Classifier on an Organized Image Set

The sample images that Create ML uses to train the custom classifier were taken with a set of categories in mind. For your own app, decide on a set of classification labels before preparing images. Take the following measures to improve your data set:

- Aim for a minimum of 10 images per category—the more, the better.
- Avoid highly unbalanced datasets by preparing a roughly equal number between categories.
- Make your model more robust by enabling the Create ML UI's Augmentation options: Crop, Rotate, Blur, Expose, Noise, and Flip.
- Include redundancy in your training set: Take lots of images at different angles, on different backgrounds, and in different lighting conditions. Simulate real-world camera capture, including noise and motion blur.
- Photograph sample objects in your hand to simulate real-world users that try to classify objects in their hands.
- Remove other objects, especially ones that you'd like to classify differently, from view.

If you're using photos taken on an iOS device to train your model, you can use the macOS utility, Image Capture, to import the images onto your computer, and do the following:

1. In Xcode, open `ImageClassifierPlayground.playground` and display the Assistant Editor.

2. Click Run on the last line of the Swift Playground; this opens the Create ML training environment.

3. Place the training images you'd like to use into named folders (such as *Agapanthus*).

4. Drag the set of folders into the Assistant Editor to perform image training.

For more information about configuring the resultant model, as well as screenshots of the Create ML UI, see `Creating an Image Classifier Model`.

Create ML exports its trained results as a `.mlmodel` file, which you can import into your app in Xcode. After importing the model, you can examine the prototypical image size by opening the model file in Xcode's navigation menu. For example, a parameter such as "Color 299 x 299" indicates the size of the training image. You can also confirm the size of the model.

Build an iOS App Around the Classifier

The app leverages the trained model and uses Vision for both registration and classification:

- It performs registration on subsequent video frame buffers to deem when the user is still enough for image capture.

- When the user is holding the camera sufficiently still, it performs image classification on the frame, attempting to identify the focused object as one of the categories in the Create ML classifier.

- If the confidence score associated with a classifier exceeds a high confidence threshold of `0.9`, the app shows its most confident classification through an overlay.

Use Registration for Scene Stability

Registration takes and aligns two images to determine the relative difference. Vision's registration operation uses an inexpensive, fast algorithm that tells the app if the subject is still and stable. Theoretically, the app could make a classification request on every frame buffer, but classification is a computationally expensive operation—so attempting to classify every frame could result in delays and poor performance with the UI. Classify the scene in a frame only if the registration algorithm determines that the scene and camera are still, indicating the user's intent to classify an object.

The FlowerShop app uses `VNSequenceRequestHandler` with `VNTranslationalImageRegistrationRequest` objects to compare consecutive frames, keeping a history of 15 frames. This amount of history amounts to half a second of capture at 30 frames per second and carries no special significance beyond empirical tuning. It takes the result of a request as `alignmentObservation.alignmentTransform` to determine if the scene is stable enough to perform classification. Check for scene stability by performing a request on the sequence request handler:

```
let registrationRequest = VNTranslationalImageRegistrationRequest(targetedCVPixelBuf
```

This algorithm deems a scene to be stable if the Manhattan distance between frames is less than 20:

```
fileprivate func sceneStabilityAchieved() -> Bool {
    // Determine if we have enough evidence of stability.
    if transpositionHistoryPoints.count == maximumHistoryLength {
        // Calculate the moving average.
        var movingAverage: CGPoint = CGPoint.zero
        for currentPoint in transpositionHistoryPoints {
            movingAverage.x += currentPoint.x
            movingAverage.y += currentPoint.y
        }
        let distance = abs(movingAverage.x) + abs(movingAverage.y)
        if distance < 20 {
            return true
        }
    }
    return false
}
```

After registration has determined that the scene is longer varying, the app sends the stable frame to Vision for Core ML classification:

```
if self.sceneStabilityAchieved() {
    showDetectionOverlay(true)
    if currentlyAnalyzedPixelBuffer == nil {
        // Retain the image buffer for Vision processing.
        currentlyAnalyzedPixelBuffer = pixelBuffer
        analyzeCurrentImage()
    }
} else {
    showDetectionOverlay(false)
}
```

Perform Image Classification

The sample app wraps two request objects—a barcode detection request and an image classification request—in a single request execution so Vision can perform them together.

Performing the combined request is faster than performing separate requests, since Vision can share the same visual data between both.

Classification contains a setup stage and a performance stage. The setup stage involves initializing requests for the types of objects you'd like Vision to detect and defining completion handlers to tell the app how to handle detection results after the requests finish their work.

The sample code sets up both a classification request and a barcode detection request. FlowerShop uses barcode identification to label an object—fertilizer—for which it has no training data. For example, the curator of a museum exhibit or owner of a flower shop can place the barcode beside or in place of an actual item, so that scanning the barcode classifies the item.

By using it as a proxy for the actual item, the app can still provide a confident classification even if the user doesn't scan the actual item. This kind of proxy works particularly well for items that Create ML may have trouble training through images, such as fertilizer, gasoline, transparent gases, and clear liquids. Set up this kind of barcode detection using a `VNDetectBarcodesRequest` object:

```
let barcodeDetection = VNDetectBarcodesRequest(completionHandler: { (request, error)
    if let results = request.results as? [VNBarcodeObservation] {
        if let mainBarcode = results.first {
            if let payloadString = mainBarcode.payloadStringValue {
                self.showProductInfo(payloadString)
            }
        }
    }
})
self.analysisRequests = ([barcodeDetection])

// Setup a classification request.
guard let modelURL = Bundle.main.url(forResource: "FlowerShop", withExtension: "mlmo
    return NSError(domain: "VisionViewController", code: -1, userInfo: [NSLocalizedD
}
guard let objectRecognition = createClassificationRequest(modelURL: modelURL) else {
    return NSError(domain: "VisionViewController", code: -1, userInfo: [NSLocalizedD
}
self.analysisRequests.append(objectRecognition)
```

The sample appends the normal model-based classification request to the same array. You can create both requests at once, but the sample code staggers the classification request to guard against failure to load the Core ML model. The classification request loads the Core ML classifier into a `VNCoreMLRequest` object:

```
let objectClassifier = try VNCoreMLModel(for: MLModel(contentsOf: modelURL))
let classificationRequest = VNCoreMLRequest(model: objectClassifier, completionHandl
```

Defining the requests and completion handlers concludes the setup stage; the second stage performs identification in real time. The sample sends the stable frame to the classifier and tells Vision to perform classification by calling `perform`:

```
private func analyzeCurrentImage() {
    // Most computer vision tasks are not rotation-agnostic, so it is important to p
    let orientation = exifOrientationFromDeviceOrientation()

    let requestHandler = VNImageRequestHandler(cvPixelBuffer: currentlyAnalyzedPixel
    visionQueue.async {
        do {
            // Release the pixel buffer when done, allowing the next buffer to be pr
            defer { self.currentlyAnalyzedPixelBuffer = nil }
            try requestHandler.perform(self.analysisRequests)
        } catch {
            print("Error: Vision request failed with error \"\(error)\"")
        }
    }
}
```

Perform tasks asynchronously on a background queue, so the camera and user interface can keep running unhindered. Don't continuously queue up every buffer that the camera provides; instead, drop buffers to keep the pipeline moving. The app works with a queue of one buffer, skipping subsequent frames so long as it is still processing that buffer. When one request finishes, it queues the next buffer and submits a classification request.

```
private let visionQueue = DispatchQueue(label: "com.example.apple-samplecode.FlowerS
```

Even if captured frames don't match the size of the image under which you trained the Create ML model (299 × 299), the Vision framework crops and scales down its input images to match the model's expected size on your behalf.

Interpret Classification Results

Check the results in the request's completion handler. When you create and pass in a request, you handle results and errors and show the classification results in your app's UI.

```
if let results = request.results as? [VNClassificationObservation] {
    print("\(results.first!.identifier) : \(results.first!.confidence)")
    if results.first!.confidence > 0.9 {
        self.showProductInfo(results.first!.identifier)
    }
}
```

The sample app sets a confidence threshold of `0.9`—empirically tuned—to filter out false classifications. A score of `1.0` means only that the photo submitted for request satisfies the algorithm and trained classifier. The algorithm could output a score of `1.0` even when the classfication is wrong. When tuning your application for the optimal confidence threshold, use the output streamed to Xcode's debugger window to gauge typical confidence values, making sure to note how far the confidence spikes on typical correct classifications. A white background with no object can still yield a confidence score of `0.6`.

The sample shows the top result, but in a search app, you can rank the labels by confidence, from most confident classification to least. The array of confidence scores and classifications is available, so use more than the top result if it fits your app's context. Try different thresholds to determine the best balance of reducing false positives and surfacing real-world results when they are correct; a result can be correct at a lower confidence score, like `0.8`. Even though this app's threshold is `0.9`, the ideal threshold may vary from model to model.

Release Your Buffers

After processing your buffers, be sure to release them to prevent them from queuing up. Because the input is a capture device that is constantly streaming frames, your app will run out of memory quickly if you don't discard extra frames. The sample app limits the number of queued frame buffers to only one, which prevents overflow from happening and clears the buffer by setting it to `nil`:

```
self.currentlyAnalyzedPixelBuffer = nil
```

See Also

Machine learning image analysis

{} Classifying Images with Vision and Core ML

Crop and scale photos using the Vision framework and classify them with a Core ML model.

`class VNCoreMLRequest`

An image-analysis request that uses a Core ML model to process images.

`class VNClassificationObservation`

An object that represents classification information that an image-analysis request produces.

`class VNPixelBufferObservation`

An object that represents an image that an image-analysis request produces.

`class VNCoreMLFeatureValueObservation`

An object that represents a collection of key-value information that a Core ML image-analysis request produces.