

Machine learning in iOS.

Week 4. HealthySnacks.

Build an application that will distinguish between healthy snacks and unhealthy ones.

Input: Photos in the iPhone gallery or a photo taken with the phone camera.

Output: The label whether the snack is healthy or not and some estimation - let's call it probability - how the system judges the correct classification of the snack, how confident the system is in its decision.

Two frameworks should be used: **Core ML** and **Vision**.

Open HealthySnacks starter app and analyze it.

In order to do machine learning on the device, you need to have a trained model. In this application, we will use a model that has already been created and trained for you.

The model is trained to recognize the following snacks:

HEALTHY	UNHEALTHY
Apple	Cake
Banana	Candy
Carrot	Cookie
Grape	Doughnut
Juice	Hot Dog
Orange	Ice Cream
Pineapple	Muffin
Salad	Popcorn
Strawberry	Pretzel
Watermelon	Waffle

What the model actually predicts is not just a label (“healthy” or “unhealthy”) but a probability distribution, where each classification is given a probability value, like:


[0.15, 0.85]

which can be interpreted as 15% being healthy and 85% being unhealthy.

Core ML.

Xcode will automatically write most of the code for you after you add a model to the app.


With the HealthySnacks project open in Xcode, drag the HealthySnacks.mlmodel file into the project to add it to the app (or use File -> Add Files).

**HealthySnacks**

Type Neural Network Classifier

Size 5 MB

Availability iOS 11.0+ | macOS 10.13+ | tvOS 11.0+ | watchOS 4.0+

Model Class  HealthySnacks
Automatically generated Swift model class

Metadata

Preview

Predictions

Utilities

Metadata

Description

Image classifier (squeezenet_v1.1) created by Turi Create (version 4.1.1)

Author

—

License

—

Version

—

Additional Metadata

features

image

max_iterations

200

model

squeezenet_v1.1

target

label

turicreate_version

4.1.1

type

ImageClassifier

Class Labels

2

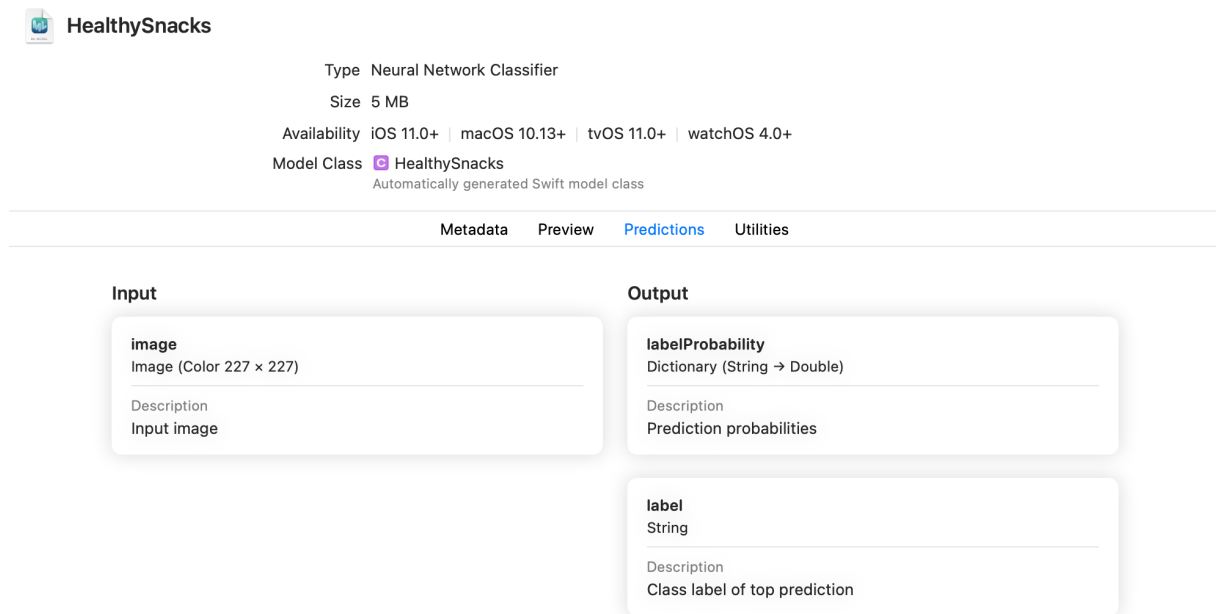
Labels

healthy

unhealthy

Layer Distribution

Layer	Count
ActivationReLU	26
Convolution	26
Concat	8
PoolingMax	3
InnerProduct	1
PoolingAverage	1
Softmax	1
Flatten	1



Any image you pick from the photo library or take with the camera must be resized to 227×227 before you can use it with this Core ML model.

The reason for this restriction is that the SqueezeNet architecture expects an image of exactly this size.

For a single 227×227 image, SqueezeNet performs 390 million calculations.

Making the image smaller will make the model faster, and it can even help the models learn better since scaling down the image helps to remove unnecessary details that would otherwise just confuse the model.

The HealthySnacks model has two outputs. It puts the probability distribution into a dictionary named `labelProbability` that will look like this:

```
labelProbability = [ "healthy": 0.15, "unhealthy": 0.85 ]
```

For convenience, the second output from the model is the class label of the top prediction: "healthy" if the probability of the snack being healthy is greater than 50%, "unhealthy" if it's less than 50%.

Press Command-B (or choose Product->Build) and then click on Model Class HealthySnacks. A window with the code of the class will open.

You're NOT going to use the automatically generated (from the model) Swift classes to make the predictions.

There are a few reasons for this, most importantly that the images need to be scaled to 227×227 pixels and placed into a `CVPixelBuffer` object before you can call the prediction method.

So, you're going to be using yet another framework: **Vision**.

Vision.

Classifying Images with Vision and Core ML:

https://developer.apple.com/documentation/coreml/model_integration_samples/classifying_images_with_vision_and_core_ml

Vision helps with computer vision tasks. For example, it can detect rectangular shapes and text in images, detect faces and even track moving objects.

Most importantly for you, Vision makes it easy to run Core ML models that take images as input.

You can also combine Vision tasks into an efficient image-processing pipeline. For example, in an app that detects people's emotions, you can build a Vision pipeline that first detects a face in the image and then runs a Core ML-based classifier on just that face to see whether the person is smiling or frowning.

It's recommended that you use Vision to drive Core ML if you're working with images.

Vision also performs a few other tasks, such as rotating the image so that it's always right-size up and matching the image's color to the device's color space.

The way Vision works is that you create a **VNRequest** object, which describes the task you want to perform, and then you use a **VNImageRequestHandler** to execute the request. Since you'll use Vision to run a Core ML model, the request is a subclass named **VNCoreMLRequest**.

Creating the VNCoreMLRequest.

ViewControllerSwift:

Add this code:

```
import CoreML
import Vision
```

Add the following code inside the ViewController class below the @IBOutlets:

```
lazy var classificationRequest: VNCoreMLRequest = {
    do {
        let healthySnacks = try HealthySnacks(configuration:
                                MLModelConfiguration())
        let visionModel = try VNCoreMLModel(
            for: healthySnacks.model)
        let request = VNCoreMLRequest(model: visionModel,
            completionHandler: {
```

```

        [weak self] request, error in
            print("Request is finished!", request.results ?? " Unknown error.")
    })
    request.imageCropAndScaleOption = .centerCrop
    return request
} catch {
    fatalError("Failed to create VNCoreMLModel: \(error)")
}
}()

```

Vision requests run asynchronously so you can supply a completion handler that will receive the results.

The `imageCropAndScaleOption` tells Vision how it should resize the photo down to the 227×227 pixels that the model expects.

You should set the request's `imageCropAndScaleOption` property so that it uses the same method that was used during training.

Vision offers three possible choices:

- `centerCrop` – this option first resizes the image so that the smallest side is 227 pixels, and then it crops out the center square
- `scaleFill` – the image gets resized to 227×227 without removing anything from the sides, so it keeps all the information from the original image — but if the original wasn't square then the image gets squashed
- `scaleFit` – keeps the aspect ratio intact but compensates by filling in the rest with black pixels.

Performing the request.

```

func classify(image: UIImage) {
    guard let ciImage = CIImage(image: image) else {
        print("Unable to create CIImage")
        return
    }
    let orientation = CGImagePropertyOrientation(
        image.imageOrientation)
    DispatchQueue.global(qos: .userInitiated).async {
        let handler = VNImageRequestHandler(
            ciImage: ciImage,
            orientation: orientation)
        do {
            try handler.perform([self.classificationRequest])
        } catch {
            print("Failed to perform classification: \(error)")
        }
    }
}

```

The `UIImage` has an `imageOrientation` property that describes which way is up when the image is to be drawn. For example, if the orientation is “down,” then the image should be rotated 180 degrees. You need to tell Vision about the image's orientation so that it can rotate the image if necessary, since Core ML expects images to be upright.

Create a new `VNImageRequestHandler` for this image and its orientation information, then call `perform` to actually do execute the request.

The Core ML model does not take “image orientation” as an input.

This is why you need to tell Vision about the image’s orientation so that it can properly rotate the image’s pixels before they get passed to Core ML. Since Vision uses `CGImage` or `CImage` instead of `UIImage`, you need to convert the `UIImage.Orientation` value to a `CGImagePropertyOrientation` value.

Run the app and note its results in the debug pane.

Change the handler which displays the results to this:

```
self?.processObservations(for: request, error: error)
```

Add the `processObservations` method:

```
func processObservations(
    for request: VNRequest,
    error: Error?) {
    DispatchQueue.main.async {
        if let results = request.results
            as? [VNClassificationObservation] {
            if results.isEmpty {
                self.resultsLabel.text = "nothing found"
            } else {
                self.resultsLabel.text = String(
                    format: "%@ %.1f%%",
                    results[0].identifier,
                    results[0].confidence * 100)
            }
        } else if let error = error {
            self.resultsLabel.text =
                "error: \(error.localizedDescription)"
        } else {
            self.resultsLabel.text = "???"
        }
        self.showResultsView()
    }
}
```

When you viewed the Core ML model in Xcode (by selecting the `.mlmodel` file in the Project navigator), it said that the model had two outputs: a dictionary containing the probabilities and the label for the top prediction. However, the Vision request gives you an array of `VNClassificationObservation` objects instead. Vision takes that dictionary from Core ML and turns it into its own kind of “observation” objects.

You can add the following clause to the `if` statement:

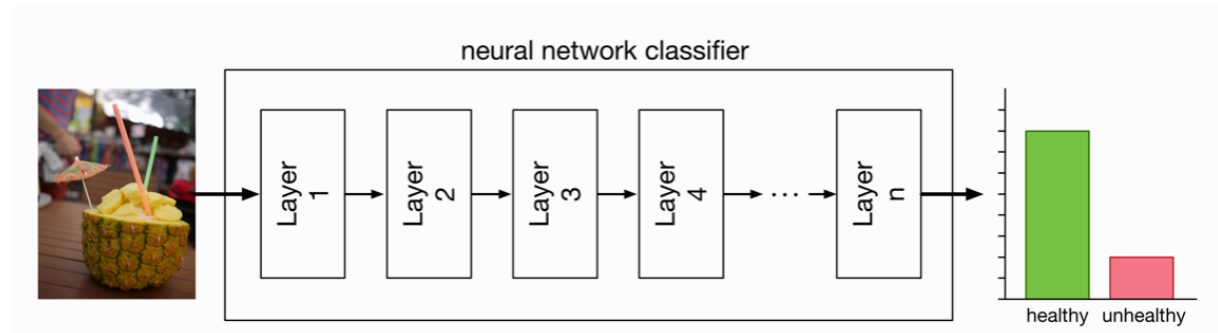
```
if results.isEmpty {
    ...
} else if results[0].confidence < 0.8 {
    self.resultsLabel.text = "not sure"
```

```
} else {  
  ...
```

How does the classifier work?

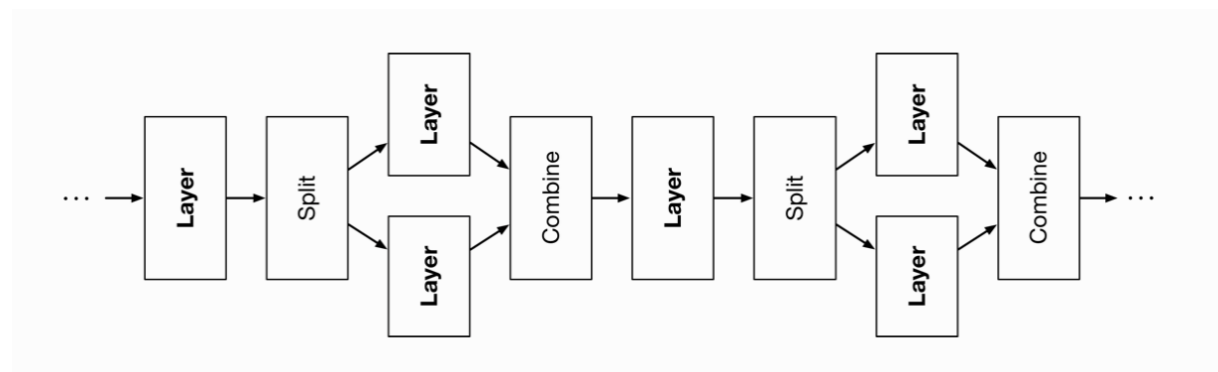
An example from the book.

The HealthySnacks.mlmodel is a neural network classifier.



You can think of a neural network as a pipeline that transforms data in several different stages.

The SqueezeNet neural network architecture that the HealthySnacks model is based on looks something like this:



SqueezeNet has 67 layers.

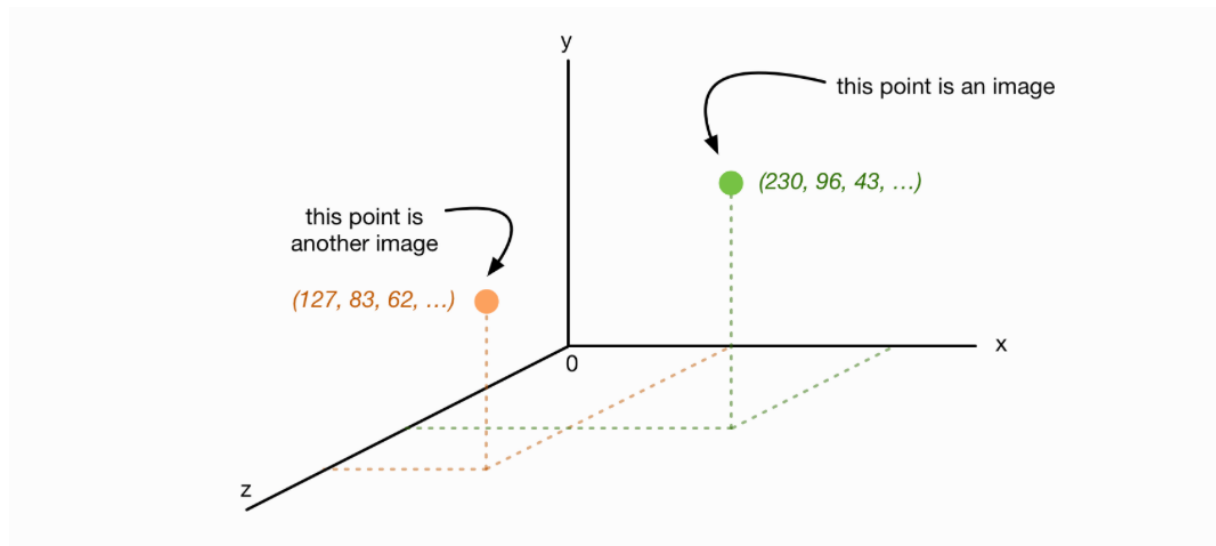
Neural networks, like most machine learning models, can only work with numerical data. The input image and the output probabilities — are all represented as numbers already.

The input image is 227×227 pixels and is a color image, so you need $227 \times 227 \times 3 = 154,587$ numbers to describe an input image. For the sake of explanation, let's round this down to 150,000 numbers.

Each pixel needs three numbers because color is stored as RGB: a red, green and blue intensity value. Some images also have a fourth channel, the alpha channel, that stores

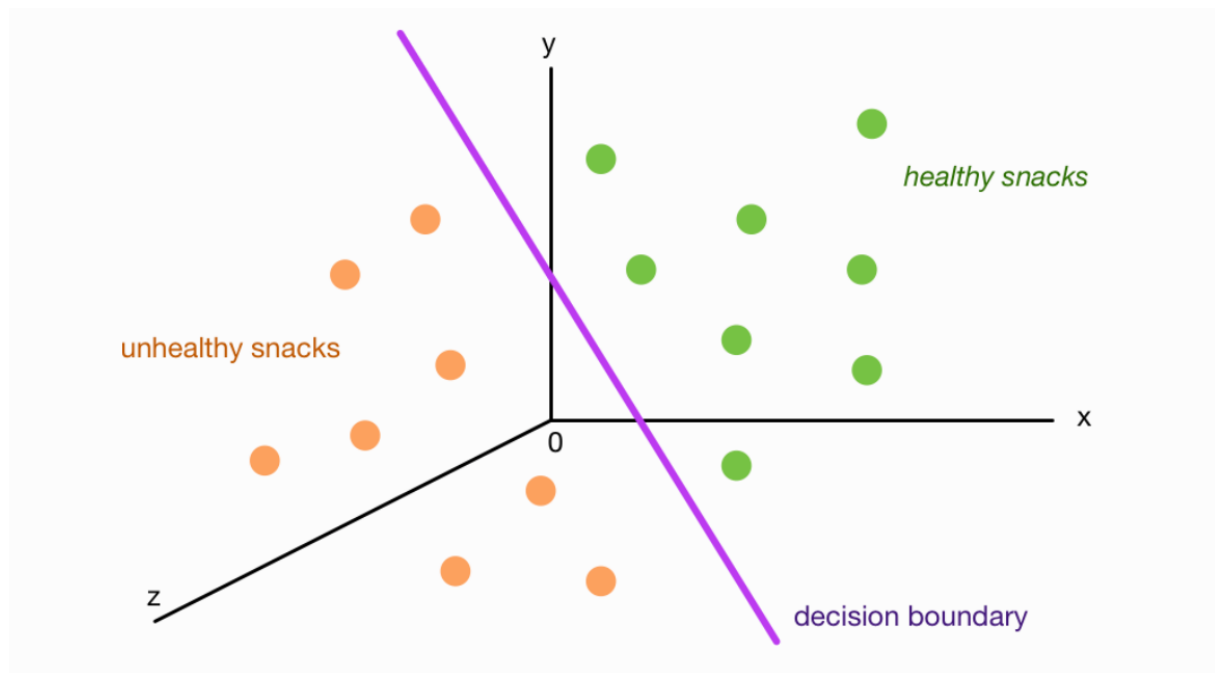
transparency information, but this is typically not used by image classifiers. It's OK to use an RGBA image as input, but the classifier will simply ignore the alpha value.

The big idea: **Each of the 227×227 input images can be represented by a unique point in a 150,000-dimensional space.**



Given 3 numbers (x, y, z) you can describe any point in 3-dimensional space. Given 150,000 numbers with the RGB values of all the pixels in the image, you end up at a point in 150,000-dimensional space.

To classify the images, you want to be able to draw a hyperplane through this high-dimensional space and say, "All the images containing healthy snacks are on this side of the hyperplane, and all the images with unhealthy snacks are on the other side." If that would be possible, then classifying an image is easy: You just have to look at which side of the line the image's point falls.



This line is called the decision boundary. It's the job of the classifier model to learn where that decision boundary lies.

The problem is that you cannot draw a nice hyperplane through the 150,000-dimensional pixel space because ordering the images by their pixel values means that the healthy and unhealthy images are all over the place.

Since pixels capture light intensity, images that have the same color and brightness are grouped together, while images that have different colors are farther apart. Apples can be red or green but, in pixel space, such images are not close together. Candy can also be red or green, so you'll find pictures of apples mixed up with pictures of candy.

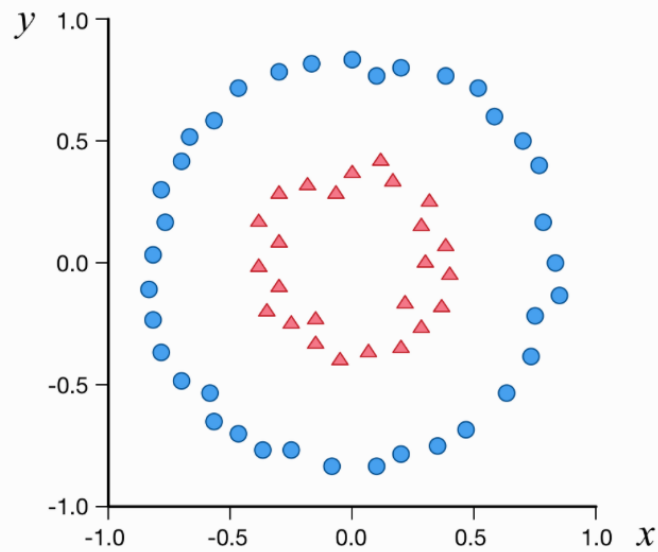
You cannot just look at how red or green something is to decide whether this image contains something healthy or unhealthy.

All the information you need to make a classification is obviously contained in the images, but the way the images are spread out over this 150,000-dimensional pixel space is not very useful. What you want instead is a space where all the healthy snacks are grouped together, and all the unhealthy snacks are grouped together, too.

This is where the neural network comes in: **The transformations that it performs in each stage of the pipeline will twist, turn, pull and stretch this coordinate space, until all the points that represent healthy snacks will be on one side and all the points for unhealthy snacks will be on the other**, and you can finally draw that hyperplane between them.

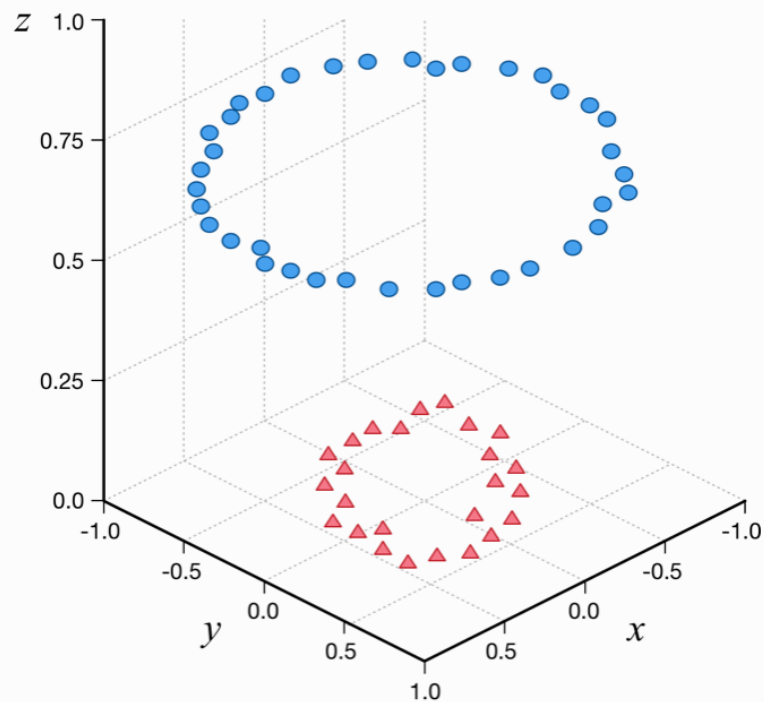
[A concrete example from the book.](#)

Here is a famous example that should illustrate the idea. In this example the data is two-dimensional, so each input consists of only two numbers (x , y). This is also a binary classification problem, but in the original coordinate space it's impossible to draw a straight line between the two classes:



In theory, you could classify this dataset by learning to separate this space using an ellipse instead of a straight line, but that's rather complicated. It's much easier to perform a smart transformation that turns the 2D space into a 3D space by giving all points a z -coordinate too. The points from class A (the triangles) get a small z value, the points from class B (the circles) get a larger z value.

Now the picture looks like this:



After applying this transformation, both classes get cleanly separated. You can easily draw a hyperplane between them at $z = 0.5$. Any point with z -coordinate less than 0.5 belongs to class A, and any point with z greater than 0.5 belongs to class B.

The closer a point's z -coordinate is to the hyperplane, the less confident the model is about the class for that point. This also explains why probabilities get closer to 50% when the HealthySnacks model can't decide whether the snack in the image is healthy or unhealthy. In that case, the image gets transformed to a point that is near the decision boundary. Usually, the decision boundary is a little fuzzy and points with z close to 0.5 could belong to either class A (triangles) or class B (circles).

Neural networks can automatically learn to make these kinds of transformations, to convert the input data from a coordinate space where it's hard to tell the points apart, into a coordinate space where it's easy. **That is exactly what happens when the model is trained. It learns the transformations and how to find the best decision boundary.**

To classify a new image, the neural network will apply all the transformations it has learned during training, and then it looks at which side of the line the transformed image falls.

In general, the more complex the data, the deeper the neural network has to be.

For the 2D example above, a neural net with just two layers will suffice. For images, which are clearly much more complex, the neural net needs to be deeper because it needs to perform more transformations to get a nice, clean decision boundary.

Multi-class classification.

Open MultiSnacks starter app.

Drag (or add file) the MultiSnacks.mlmodel from the downloaded resources into the Xcode project.

Look at the new model in XCode.

Change the function processObservations(for: error:) to:

```
if results.isEmpty {
    self.resultsLabel.text = "nothing found"
} else {
    let top3 = results.prefix(3).map { observation in
        String(format: "%@ %.1f%%", observation.identifier,
            observation.confidence * 100)
    }
    self.resultsLabel.text = top3.joined(separator: "\n")
}
```

Using CoreML without Vision.

Use the starter project HealthySnacks again and add the HealthySnacks.mlmodel to the project.

Open HealthySnacks.swift (generated from the model file) and analyze it.

Note two prediction methods.

The names that Xcode generates for parameters depend on the names of the inputs and outputs in the .mlmodel file. For this particular model, the input is called “image” and so the method becomes prediction(image:). If the input were called something else in the .mlmodel file, such as “data,” then the method would be prediction(data:). The same is true for the names of the outputs in the HealthySnacksOutput class. This is something to be aware of when you’re importing a Core ML model: different models will have different names for the inputs and outputs — another thing you don’t have to worry about when using Vision.

The HealthySnacksOutput class provides two features containing the outputs of the model: a dictionary called labelProbability and a string called simply label. The label is simply the name of the class with the highest probability and is provided for convenience. The dictionary contains the names of the classes and the confidence score for each class, so it’s the same as the probability distribution but in the form of a dictionary instead of an array. The difference with Vision’s array of VNClassificationObservation objects is that the dictionary is not sorted.

In order to use the HealthySnacks class without Vision, you have to call its prediction(image:) method and give it a CVPixelBuffer containing the image to classify. When the prediction method is done it returns the classification result as a HealthySnacksOutput object.

Add the following property to ViewController to create an instance of the model:

```
let healthySnacks = HealthySnacks()
```

Now, you need a way to convert the UIImage from UIImagePickerController into a CVPixelBuffer.

```
func pixelBuffer(for image: UIImage) -> CVPixelBuffer? {
    let model = healthySnacks.model

    let imageConstraint = model.modelDescription
        .inputDescriptionsByName["image"]!
        .imageConstraint!

    let imageOptions: [MLFeatureValue.ImageOption: Any] = [
        .cropAndScale: VNImageCropAndScaleOption.scaleFill.rawValue
    ]

    return try? MLFeatureValue(
        cgImage: image.cgImage!,
        constraint: imageConstraint,
        options: imageOptions).imageBufferValue
}
```

The constraint is an `MMLImageConstraint` object that describes the image size that is expected by the model input. The options dictionary lets you specify the how the image gets resized and cropped.

Change the `classify(image:)` method to the following:

```
func classify(image: UIImage) {
    DispatchQueue.global(qos: .userInitiated).async {

        if let pixelBuffer = self.pixelBuffer(for: image) {

            if let prediction = try? self.healthySnacks.prediction(
                image: pixelBuffer) {

                let results = self.top(1, prediction.labelProbability)
                self.processObservations(results: results)
            } else {
                self.processObservations(results: [])
            }
        }
    }
}
```

Where:

```
func top(_ k: Int, _ prob: [String: Double])
-> [(String, Double)] {
    return Array(prob.sorted { $0.value > $1.value }
        .prefix(min(k, prob.count)))
}
```

Change function `processObservations` to:

```
func processObservations(
    results: [(identifier: String, confidence: Double)]) {
    DispatchQueue.main.async {
        if results.isEmpty {
            self.resultsLabel.text = "nothing found"
        } else if results[0].confidence < 0.8 {
            self.resultsLabel.text = "not sure"
        } else {
            self.resultsLabel.text = String(
                format: "%@ %.1f%%",
                results[0].identifier,
                results[0].confidence * 100)
        }
        self.showResultsView()
    }
}
```

Challenge.

Apple provides a number of Core ML models that you can download for free, from <https://developer.apple.com/machine-learning/models/>.

Download the MobileNetV2 model and add it to the app.

This model is similar to the classifier you implemented, (which is based on SqueezeNet). The main difference is that HealthySnacks is trained to classify 20 different snacks into two groups: healthy or unhealthy. The MobileNetV2 model from Apple is trained to understand 1,000 classes of different objects (it's a multi-class classifier).

Try to add this new model to the app.

It should only take a small modification to make this work.