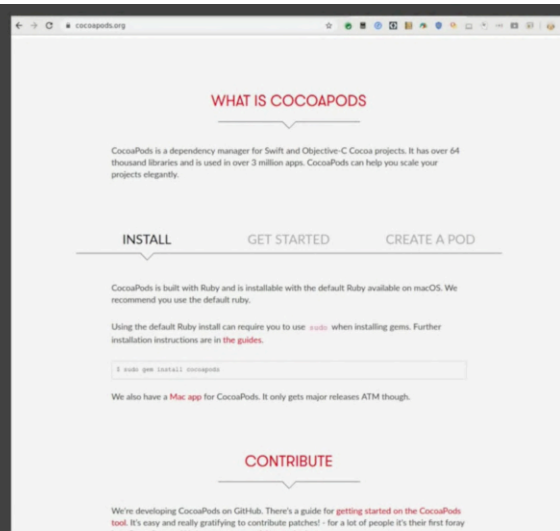
Overview of TensorFlowLiteSwift

- Swift library to run TensorFlowLite models on an iOS device.
- Current version is 0.2.0

https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/experimental/swift

In order to use TensorFlow Lite with Swift, there's a TensorFlow Lite Swift pod which gives you the interpreter and the various

```
$> sudo gem install cocoapods
```



To add TensorFlow Lite to your iOS project, you use a pod file. If you're not familiar with this, the technology is called CocoaPods. It gives you an easy way to add dependencies to your projects. Install it with the command sudo gem install Cocoapods on your Mac.

Podfile

Podfile

```
# Pods for 'Your Project'
pod 'TensorFlowLiteSwift'
```

Install Command

```
$> cd /path/to/directory/containing/podfile
$> pod install
```

Then in Your Project directory, you put a file called pod file with no extension. Then you add the text pod TensorflowLiteSwift. Go to the directory and issue the command pod install. This will add the dependencies and create a new.xcworkspace folder, and you'll work with that going forward.

Getting the Model

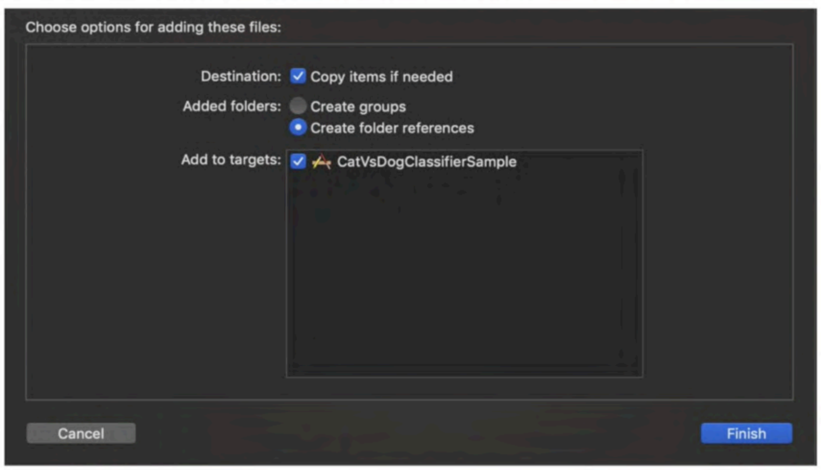
- Python notebook to train the model:

  bit.ly/makecatsdogs

- After running all the cells, you get model(.tflite) and labels(.txt) files

For the cats versus dogs project, you'll need a model and a set of labels. I've included them in the open source repository for this project, but if you need new ones, you can always run the notebook at this URL. Make sure you download the.tflite and labels file when you're done.

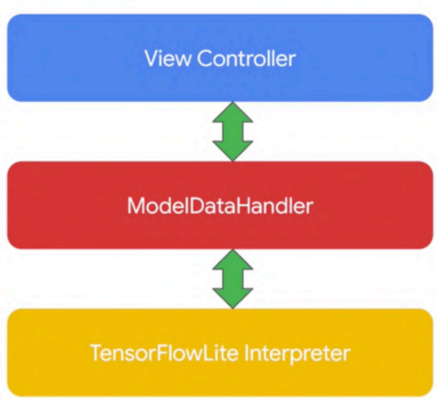
Adding Model and Labels

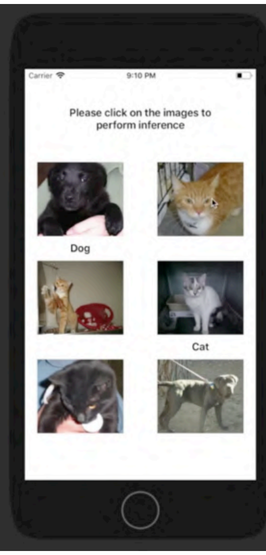


When adding models and labels to your application, you can just drag them to your project folder in Xcode

One thing to note, however, is that you'll need to check the copy items if needed, checkbox and Xcode. This ensures that the model and labels files are deployed to the device or emulator when you run.

App Architecture





Python notebook to train the model for the cats Vs dog project :

https://github.com/lmoroney/dlaicourse/blob/master/TensorFlow%20Deployment/Course%202%20-%20TensorFlow%20Lite/Week%201/Examples/TFLite_Week1_Transfer_Learning.ipynb

Interpreter

- Performs the inference using the Tflite model
- Input is passed into the input tensors
- Resulting inferences are available in the output tensors

You can think of the interpreter as the main engine that drives the inference process. We have to copy our input image pixel buffer values to the input of the interpreter and invoke the methods responsible for performing inference. After the inferences is performed, the results will be available in the output tensors of the interpreter.

Steps Involved in Performing Inference

Remember the interpreter is added to your app using the pod file we discussed earlier. If you're building an app, you'll need to have done that on none of the subsequent code will work.

1 — 2 — 3 — 4

1. Initialize the Interpreter

Model is loaded in the interpreter at this stage

2. Preparing the Image Input

Input image pixel buffer is converted to the format recognized by the model

3. Perform Inference

Pass input to the Interpreter and Invoke the Interpreter

4. Obtain and Map Results

Map our resulting confidence values to labels

ModelDataHandler.swift

```swift
let modelPath = Bundle.main.path(
    forResource: modelFilename, ofType: modelFileInfo.extension)

var options = InterpreterOptions()
options.threadCount = threadCount

interpreter = try Interpreter(modelPath: modelPath,
                              options: options)
```

You need to initialize the interpreter. You begin by creating a reference to the model file. If you copied it as shown earlier, it will be in the app bundles main path. So you can point the model path variable at bundle.main.path, specifying the file name and type.

ModelDataHandler.swift

```swift
let modelPath = Bundle.main.path(
    forResource: modelFilename, ofType: modelFileInfo.extension)

var options = InterpreterOptions()
options.threadCount = threadCount

interpreter = try Interpreter(modelPath: modelPath,
                              options: options)
```

ModelDataHandler.swift

```swift
let modelPath = Bundle.main.path(
    forResource: modelFilename, ofType: modelFileInfo.extension)

var options = InterpreterOptions()
options.threadCount = threadCount

interpreter = try Interpreter(modelPath: modelPath,
                              options: options)
```

Then you simply instantiate the interpreter, passing it the path of the model and the options that you want to use.

ModelDataHandler.swift

Next up, you'll want to allocate input tensors to reserve memory for them. It's a straightforward as calling the allocate tensors method on the initialized interpreter.

```
do {
    try interpreter.allocateTensors()
}
catch let error {
}
```

Steps Involved in Performing Inference

1. **Initialize the Interpreter**
   Model is loaded in the interpreter at this stage
2. **Preparing the Image Input**
   Input image pixel buffer is converted to the format recognized by the model
3. **Perform Inference**
   Pass input to the Interpreter and Invoke the Interpreter
4. **Obtain and Map Results**
   Map our resulting confidence values to labels

Preparing the Input

- Model Expects pixel buffer of size 224 x 224 x 3
- iOS uses CVPixelBuffer to represent images in memory
- CVPixelBuffer has Alpha as well as RGB
- Need to extract R, G, B from CVPixelBuffer and normalize
- Final output has to be of type ‘Data’

Preparing the Input

- Model Expects pixel buffer of size 224 x 224 x 3
- iOS uses CVPixelBuffer to represent images in memory
- CVPixelBuffer has Alpha as l as RGB
- Need to extract R, G CVPixelBuffer and normalize
- Final output e 'Data'

https://developer.apple.com/documentation/corevideo/cvpixelbuffer-q2e

Preparing the Input

- Model Expects pixel buffer of size 224 x 224 x 3
- iOS uses CVPixelBuffer to represent images in memory
- CVPixelBuffer has Alpha as well as RGB
- Need to extract R, G, B from CVPixelBuffer and normalize
- Final output has to be of type 'Data'

Preparing the Input

- Model Expects pixel buffer of size 224 x 224 x 3
- iOS uses CVPixelBuffer to represent images in memory
- CVPixelBuffer has Alpha as well as RGB
- Need to extract R, G, B from CVPixelBuffer and normalize
- Final output has to be of type 'Data'

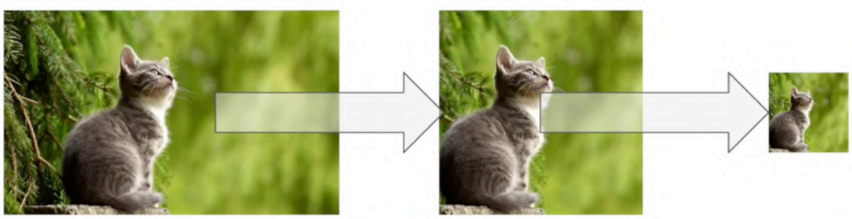
Preparing the Input

- Model Expects pixel buffer of size 224 x 224 x 3
- iOS uses CVPixelBuffer to represent images in memory
- CVPixelBuffer has Alpha as well as RGB
- Need to extract R, G, B from CVPixelBuffer and normalize
- Final output has to be of type 'Data'

https://developer.apple.com/documentation/corevideo/cvpixelbuffer-q2e

Scaling and Cropping the CVPixelBuffer

- Crop the biggest square and scale down to 224 x 224



The first step will be to get the 224 by 224 image. Probably the easiest way to do this is to find the biggest square in the image and then resize this to 224 by 224.

The Apple APIs offer V image for this. I'm not going to go into detail on these APIs, but you can learn more at this URL.

- vImage is used for image operations
- https://developer.apple.com/documentation/accelerate/vimage

ModelDataHandler.swift

The first thing is to scale the image. As we want 224 by 224, we set up a CG size object with those dimensions.

The image is represented in pixel buffer, and the project contains an extension to CV pixel buffer that implements center thumbnail to find the biggest square and crop the image to it. We'll call that the thumbnail pixel buffer.

```
let inputWidth = 224
let inputHeight = 224
let scaledSize = CGSize(width: inputWidth, height: inputHeight)

let thumbnailPixelBuffer = pixelBuffer.centerThumbnail(ofSize: scaledSize)
```

ModelDataHandler.swift

```swift
let inputChannels = 3

let rgbData = rgbDataFromBuffer(
    thumbnailPixelBuffer,
    byteCount: inputWidth * inputHeight * inputChannels
)
```

We already have our width and height. We also need to inform the API that we went three channels, as we'll set up a variable for that and we'll call RGB data from buffer, passing it the thumbnail buffer, and telling it the byte count that we want, and we'll get the RGB data back.

ModelDataHandler.swift

```swift
private func rgbDataFromBuffer(_ buffer: CVPixelBuffer,byteCount: Int) ->
Data? {

    let mutableRawPointer = CVPixelBufferGetBaseAddress(buffer)
    let count = CVPixelBufferGetDataSize(buffer)
    let bufferData = Data(bytesNoCopy: mutableRawPointer, count: count,
deallocator: .none)
    var rgbBytes = [UInt8](repeating: 0, count: byteCount)

}
```

The RGB data buffer is included in the source code in model data handler.swift. This function constructs a byte buffer of type Data from the CV pixel buffer passed as an input argument. We construct a byte buffer of type data using the base address of the CV pixel buffer.

We also have to initialize an array that holds the resulting RGB values after stripping out the alpha components. This array will have 224 by 224 by 3 values, which was the byte count we passed as an argument to the function.

ModelDataHandler.swift

```swift
for component in bufferData.enumerated() {
    let offset = component.offset
    let isAlphaComponent = (offset % alphaComponent.baseOffset) ==
                                          alphaComponent.moduloRemainder

    guard !isAlphaComponent else { continue }
    rgbBytes[index] = Float(component.element) / 255.0
    index += 1
}
```

We loop through buffer data and check if the current components is an alpha components. Alpha component values are not appended to the RGB bytes array.

Notice how we divide the pixel values by 255? This is because our model is non quantized and hence expects float inputs in the range of zero to one.

Pixel values in the original CV pixel buffer can range from 0-255, hence we divide each of these values by 255 to normalize them.

After filtering and normalization, RGB bytes will have the same pixel values of the CV pixel buffer in RGB format.

ModelDataHandler.swift

Converting to the data class type is then as easy as returning them with a buffer pointer like this.

```swift
return rgbBytes.withUnsafeBufferPointer(Data.init)
```

Steps Involved in Performing Inference

1 — 2 — 3 — 4

1. **Initialize the Interpreter**
   Model is loaded in the interpreter at this stage
2. **Preparing the Image Input**
   Input image pixel buffer is converted to the format recognized by the model
3. **Perform Inference**
   Pass input to the Interpreter and Invoke the Interpreter
4. **Obtain and Map Results**
   Map our resulting confidence values to labels

ModelDataHandler.swift

```swift
// Copy the RGB data to the input Tensor.
try interpreter.copy(rgbData, toInputAt: 0)

// Run inference by invoking the Interpreter.
try interpreter.invoke()

// Get the output Tensor to process the inference results.
outputTensor = try interpreter.output(at: 0)
```

Copy the input data that we created as RGB data to the input tensor using The Interpreter to copy method

ModelDataHandler.swift

```swift
// Copy the RGB data to the input Tensor.
try interpreter.copy(rgbData, toInputAt: 0)

// Run inference by invoking the Interpreter.
try interpreter.invoke()

// Get the output Tensor to process the inference results.
outputTensor = try interpreter.output(at: 0)
```

Invoke the Interpreter

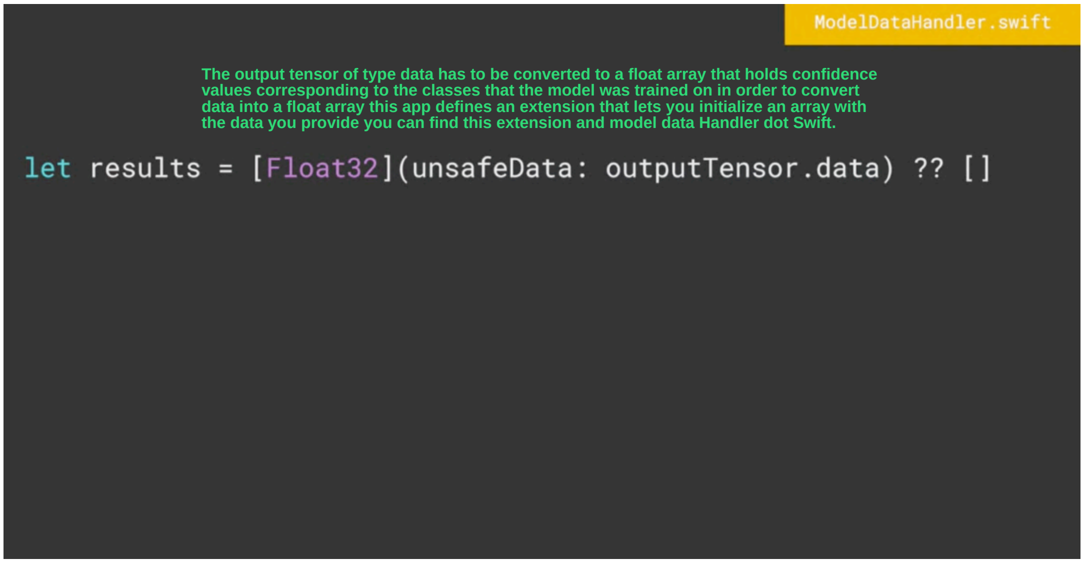
ModelDataHandler.swift

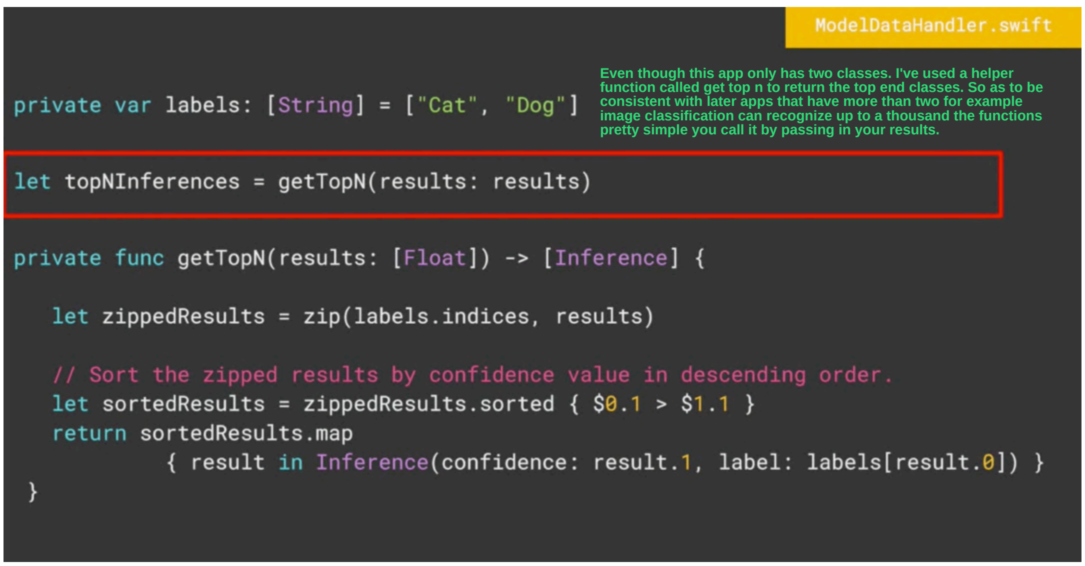
```swift
// Copy the RGB data to the input Tensor.
try interpreter.copy(rgbData, toInputAt: 0)

// Run inference by invoking the Interpreter.
try interpreter.invoke()

// Get the output Tensor to process the inference results.
outputTensor = try interpreter.output(at: 0)
```

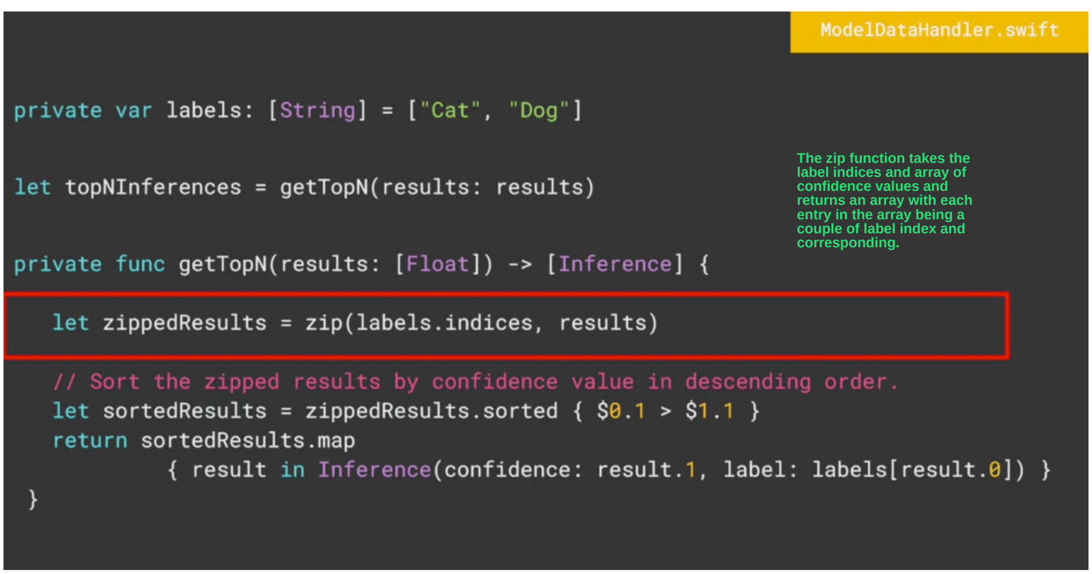
Get the output tensor which contains the inference results.

Steps Involved in Performing Inference

1. **Initialize the Interpreter** — Model is loaded in the interpreter at this stage
2. **Preparing the Image Input** — Input image pixel buffer is converted to the format recognized by the model
3. **Perform Inference** — Pass input to the Interpreter and Invoke the Interpreter
4. **Obtain and Map Results** — Map our resulting confidence values to labels


ModelDataHandler.swift
The output tensor of type data has to be converted to a float array that holds confidence values corresponding to the classes that the model was trained on in order to convert data into a float array this app defines an extension that lets you initialize an array with the data you provide you can find this extension and model data Handler dot Swift.
let results = [Float32](unsafeData: outputTensor.data) ?? []


ModelDataHandler.swift
private var labels: [String] = ["Cat", "Dog"]
Even though this app only has two classes. I've used a helper function called get top n to return the top end classes. So as to be consistent with later apps that have more than two for example image classification can recognize up to a thousand the functions pretty simple you call it by passing in your results.
let topNInferences = getTopN(results: results)
private func getTopN(results: [Float]) -> [Inference] {
let zippedResults = zip(labels.indices, results)
// Sort the zipped results by confidence value in descending order.
let sortedResults = zippedResults.sorted { $0.1 > $1.1 }
return sortedResults.map
{ result in Inference(confidence: result.1, label: labels[result.0]) }
}


ModelDataHandler.swift
private var labels: [String] = ["Cat", "Dog"]
let topNInferences = getTopN(results: results)
The zip function takes the label indices and array of confidence values and returns an array with each entry in the array being a couple of label index and corresponding.
private func getTopN(results: [Float]) -> [Inference] {
let zippedResults = zip(labels.indices, results)
// Sort the zipped results by confidence value in descending order.
let sortedResults = zippedResults.sorted { $0.1 > $1.1 }
return sortedResults.map
{ result in Inference(confidence: result.1, label: labels[result.0]) }
}

ModelDataHandler.swift

```
private var labels: [String] = ["Cat", "Dog"]

let topNInferences = getTopN(results: results)

private func getTopN(results: [Float]) -> [Inference] {

   let zippedResults = zip(labels.indices, results)

   // Sort the zipped results by confidence value in descending order.
   let sortedResults = zippedResults.sorted { $0.1 > $1.1 }
   return sortedResults.map
            { result in Inference(confidence: result.1, label: labels[result.0]) }
 }
```

Map the sorted results to an array of type inference, which is a helper structure defined in model data Handler. It holds the class name and the confidence value.

Calling Inference from UI

- ViewController uses a UICollectionView to display images
- Initializes ModelDataHandler
- Hands over Inference to ModelDatahandler

All of this needs to be called from the UI when the user touches on an image. All of that logic is in the view controller. This uses a UI collection view to display the images. It initializes the model data Handler class which contains The Interpreter and it passes the data from the image buffer to it.

ViewController.swift

Initializing the ModelDataHandler

```
private var modelDataHandler: ModelDataHandler? =
                                ModelDataHandler(
                                    modelFileInfo: MobileNet.modelInfo)
```

Viewcontroller defines a stored property that initializes the model data Handler by passing the file information for the TF light file file info is a structure defined in model data Handler that holds the name and extension of the file the initialization function performs the initialization of The Interpreter.

ViewController.swift

```swift
var result: Result?
func collectionView(_ collectionView: UICollectionView,
                    didSelectItemAt indexPath: IndexPath) {

    let image = UIImage(named: imageNames[indexPath.item])

    let pixelBuffer = pixelBuffer(from: image)

    result = modelDataHandler?.runModel(onFrame: pixelBuffer)

}
```

To pass the image to the model for inference that did select item at function on the collection view is used. This function gives an index path in the collection view that tells us the particular image that was selected

ViewController.swift

```swift
var result: Result?
func collectionView(_ collectionView: UICollectionView,
                    didSelectItemAt indexPath: IndexPath) {

    let image = UIImage(named: imageNames[indexPath.item])

    let pixelBuffer = pixelBuffer(from: image)

    result = modelDataHandler?.runModel(onFrame: pixelBuffer)

}
```

From that we can create a UI image to pull the image from the apps bundle.

ViewController.swift

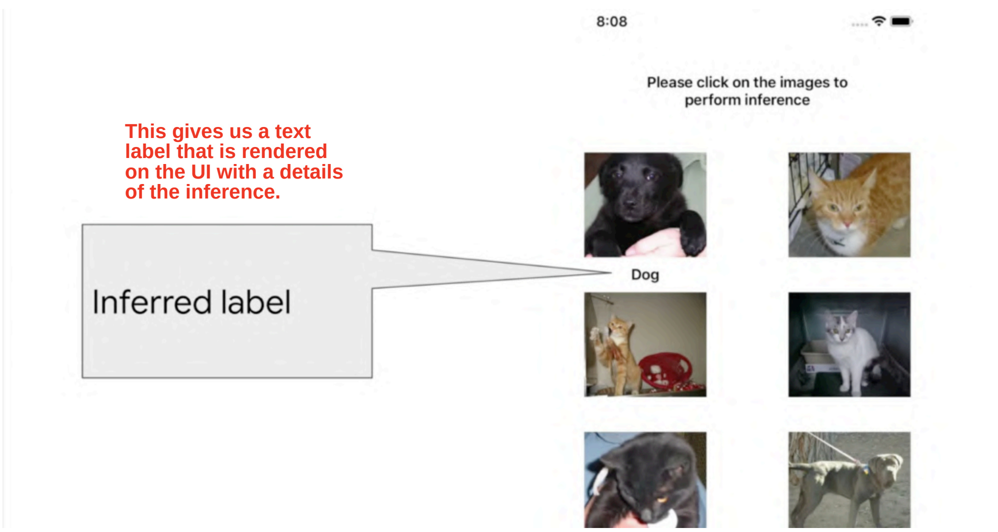
```swift
var result: Result?
func collectionView(_ collectionView: UICollectionView,
                    didSelectItemAt indexPath: IndexPath) {

    let image = UIImage(named: imageNames[indexPath.item])

    let pixelBuffer = pixelBuffer(from: image)

    result = modelDataHandler?.runModel(onFrame: pixelBuffer)

}
```

This is then used to create a pixel buffer.

ViewController.swift

```swift
var result: Result?
func collectionView(_ collectionView: UICollectionView,
                    didSelectItemAt indexPath: IndexPath) {

    let image = UIImage(named: imageNames[indexPath.item])

    let pixelBuffer = pixelBuffer(from: image)

    result = modelDataHandler?.runModel(onFrame: pixelBuffer)

}
```

The pixel buffer is passed to the model handlers run model method which converts it passes it to The Interpreter runs inference gets the results and returns that



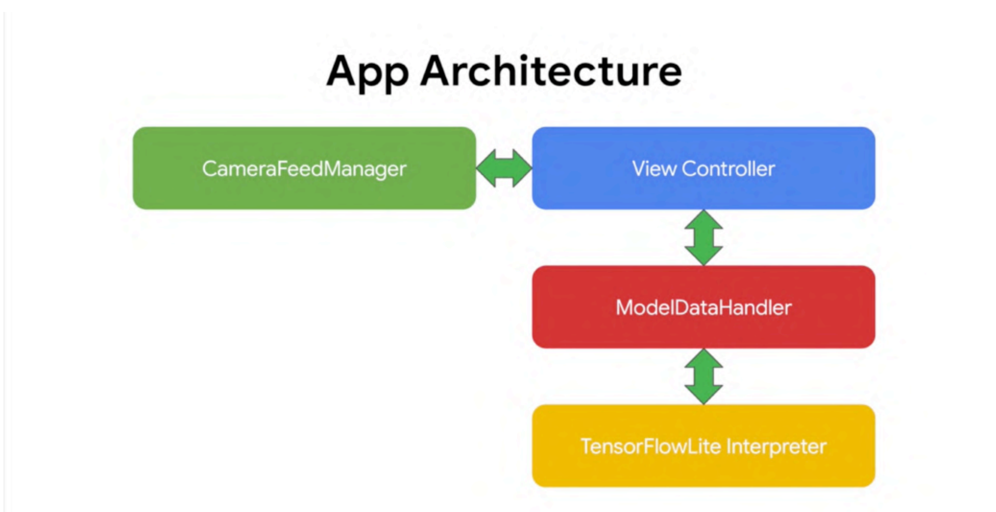


Image Classification Model Details

- Quantized MobileNet SSD trained on COCO dataset
- Trained on ImageNet 1000 classes.
- More details on the model can be found in the link below
  https://www.tensorflow.org/lite/models/image_classification/overview
- Labels file is used to list 1000 classes and map to output confidences
- You can download the .tflite file and .txt file from the following link.

Steps Involved in Performing Inference

1 — 2 — 3 — 4

1. Initialize the Interpreter
Model is loaded in the interpreter at this stage

2. Preparing the Image Input
Input image pixel buffer is converted to the format recognized by the model

3. Perform Inference
Pass input to the Interpreter and Invoke the Interpreter

4. Obtain and Map Results
Map our resulting confidence values to labels

Steps Involved in Performing Inference

1 — 2 — 3 — 4

1. Initialize the Interpreter
Model is loaded in the interpreter at this stage

2. Preparing the Image Input
Input image pixel buffer is converted to the format recognized by the model

3. Perform Inference
Pass input to the Interpreter and Invoke the Interpreter

4. Obtain and Map Results
Map our resulting confidence values to labels

ModelDataHandler.swift

```swift
let modelPath = Bundle.main.path(forResource: modelFilename,
                                 ofType: modelFileInfo.extension)

// Specify the options for the `Interpreter`.
var options = InterpreterOptions()
options.threadCount = threadCount

interpreter = try Interpreter(modelPath: modelPath, options: options)

// Load the classes listed in the labels file.
loadLabels(fileInfo: labelsFileInfo)
```

Get the model path from the bundle, then we set up our desired options. We initialize a new interpreter from these, and as our labels are in a separate file, we'll also load them in.

ModelDataHandler.swift

```swift
do {
    // Allocate memory for the model's input `Tensor`s.
    try interpreter.allocateTensors()
}
catch let error {
}
```

Allocate our tensors to make sure the memory to handle them is set up correctly

Steps Involved in Performing Inference

1. **Initialize the Interpreter** — Model is loaded in the interpreter at this stage
2. **Preparing the Image Input** — Input image pixel buffer is converted to the format recognized by the model
3. **Perform Inference** — Pass input to the Interpreter and Invoke the Interpreter
4. **Obtain and Map Results** — Map our resulting confidence values to labels

This is where we have a lot of new functionality using the Camera Feed Manager to get live video from which we'll grab frames for classification.

CameraFeedManager

- Initialized by ViewController
- Communicates with ViewController using delegates.
- Handles all camera initialization and functionality
- Uses AVFoundation to initialize and obtain frames from the back camera.
- Link to camera handling using AVFoundation
- https://developer.apple.com/documentation/avfoundation/cameras_and_media_capture/avcam_building_a_camera_app

The Camera Feed Manager handles all camera related functionality. It uses AV foundation to initialize an AV capture session that displays the incoming camera frames, on an AV capture preview layer.

As soon as Camera Feed Manager receives frames from an AV capture session, it hands over the results via delegates to the View Controller.

All the camera related classes belong in the AV foundation framework. Chances are you might be familiar with these classes.

Covering camera related functionality is not in the scope of this course, but you can check out the official Apple Developer documentation given at this link, for detailed explanation.

I will only be explaining how functions from camera feed Manager are called by the View Controller to help you understand the integration better.

ViewController.swift

```
private lazy var cameraCapture = CameraFeedManager(previewView: previewView)

override func viewWillAppear(_ animated: Bool) {

    cameraCapture.checkCameraConfigurationAndStartSession()

}

override func viewDidLoad() {
    ...
    cameraCapture.delegate = self
}
```

Of course we want to initialize the Camera Feed Manager, and we'll use a var code camera capture to keep track of it.

ViewController.swift

```
private lazy var cameraCapture = CameraFeedManager(previewView: previewView)

override func viewWillAppear(_ animated: Bool) {

    cameraCapture.checkCameraConfigurationAndStartSession()

}

override func viewDidLoad() {
    ...
    cameraCapture.delegate = self
}
```

In the view controllers, view will appear method, we will start the cameras session by calling check camera configuration and start session.

ViewController.swift

```swift
private lazy var cameraCapture = CameraFeedManager(previewView: previewView)

override func viewWillAppear(_ animated: Bool) {

    cameraCapture.checkCameraConfigurationAndStartSession()

}

override func viewDidLoad() {
    ...
    cameraCapture.delegate = self
}
```

Don't forget to set the delegates as we do here in viewDidLoad. Camera feed manager delegate has methods that are called by the camera feed Manager whenever an output frame is obtained from the AV capture session. This delegate also has methods which convey responses from AV captures session in various stages of camera handling. The app has an extension to ViewController that implements the camera feed Manager delegates to see how these delegates methods are implemented.

ViewController.swift

```swift
extension ViewController: CameraFeedManagerDelegate {
    func didOutput(pixelBuffer: CVPixelBuffer) {
        ...
        result = modelDataHandler?.runModel(onFrame: pixelBuffer)
        ...
    }
}
```

You can see the ViewController getting the pixel buffer from the camera feed Manager through the delegate and the ViewController Camera feed manager delegate extension. It's output codes, sends the pixel buffer, and note the datatype. It's a CV pixel buffer.

CameraFeedManager.swift

```swift
extension CameraFeedManager: AVCaptureVideoDataOutputSampleBufferDelegate() {
    func captureOutput(_ output: AVCaptureOutput,
                       didOutput sampleBuffer: CMSampleBuffer
                       from connection: AVCaptureConnection){
        ...
        let pixelBuffer: CVPixelBuffer? = CMSampleBufferGetImageBuffer(sampleBuffer)

        guard let imagePixelBuffer = pixelBuffer else{
            return
        }

        delegate?.didOutput(pixelBuffer: imagePixelBuffer)
    }
}
```

That is then sent from the camera feed Manager, in its AV capture video data output sample buffer delegate extension. It uses the AV capture libraries to get an image buffer, and then passes that out on the outputs.

Preparing the Input

- Expects pixel buffer of size 224 x 224 x 3
- Our CVPixelBuffer is of type BGRA_32
- Has to be converted to Pixel buffer with only R, G, B channels.
- Pixel Buffer has to be converted to Data

Scaling and Cropping the CVPixelBuffer

- Crop the biggest square and scale down to 224 x 224
- vImage is used for image operations
- https://developer.apple.com/documentation/accelerate/vimage

ModelDataHandler.swift

Handling Quantized Model Inputs

```
isModelQuantized: inputTensor.dataType == .uInt8

if isModelQuantized { return Data(bytes: rgbBytes) }
return Data(copyingBufferOf: rgbBytes.map { Float($0) / 255.0 })
```

One quick note about whether you're using quantized the non quantized models. If the model is quantized, then its input values will be integers. So if we look at the input tensor and check its datatype, we'll be able to tell what type of data the model expects. Then, depending on whether the model is quantized or not, when we create the data object we can either just copy the raw RGB bytes, or we can divide the float value by 255.

Camera Related Functions: https://developer.apple.com/documentation/avfoundation/cameras_and_media_capture/avcam_building_a_camera_app

URL for vImage: https://developer.apple.com/documentation/accelerate/vimage

Steps Involved in Performing Inference

1 — 2 — 3 — 4

1. Initialize the Interpreter
Model is loaded in the interpreter at this stage

2. Preparing the Image Input
Input image pixel buffer is converted to the format recognized by the model

3. Perform Inference
Pass input to the Interpreter and Invoke the Interpreter

4. Obtain and Map Results
Map our resulting confidence values to labels

ModelDataHandler.swift

Invoking the Interpreter

```swift
// Copy the RGB data to the input `Tensor`.
try interpreter.copy(rgbData, toInputAt: 0)

// Run inference by invoking the `Interpreter`.
try interpreter.invoke()

// Get the output `Tensor` to process the inference results.
outputTensor = try interpreter.output(at: 0)
```

Steps Involved in Performing Inference

1 — 2 — 3 — 4

1. Initialize the Interpreter
Model is loaded in the interpreter at this stage

2. Preparing the Image Input
Input image pixel buffer is converted to the format recognized by the model

3. Perform Inference
Pass input to the Interpreter and Invoke the Interpreter

4. Obtain and Map Results
Map our resulting confidence values to labels

ModelDataHandler.swift

The process is now super simple. We copy the RGB data to the input tensor, we invoke the interpreter and then we read the output tensor. We'll get an inference back. Here's where we need to understand the output and map it to the models labels.

How we handle the data from the output Tensor depends on whether the model was quantized or not. So in an effort to make the code as generic as possible, this app will handle both. It's a matter of checking the data type of the output Tensor, and if it's an Int, we execute the first branch of this case statements to read them as integers. If it's a Float, we simply copy the values out to an array type. Note that this code requires an array extension, which you can find in model datahandle.swift.

```swift
let results: [Float]
switch(outputTensor.dataType){

   case .uInt8:
        let quantization = outputTensor.quantizationParameters
        let quantizedResults = [UInt8](outputTensor.data)
        results = quantizedResults.map {
            quantization.scale * Float(Int($0) - quantization.zeroPoint)
        }

    case .float32:
        results = [Float32](unsafeData: outputTensor.data) ?? []
```

ModelDataHandler.swift

Now, finding the top values uses exactly the same code as before, where we organize our results into an array containing the label name and the confidence value, and then sort that to return the top end values.

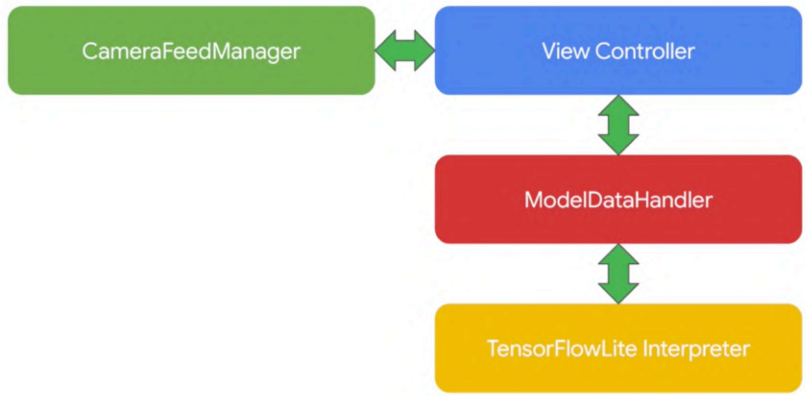
```swift
let topNInferences = getTopN(results: results)


private func getTopN(results: [Float]) -> [Inference] {

   let zippedResults = zip(labels.indices, results)

   // Sort the zipped results by confidence value in descending order.
   let sortedResults = zippedResults.sorted { $0.1 > $1.1 }.prefix(resultCount)

   return sortedResults.map { result in Inference(confidence: result.1,
                                                  label: labels[result.0]) }
 }
```

ViewController.swift

We finally display our results in a label on screen, and in this example, we've mapped the inference into an array of string suitable for display, and then join them by new lines for displaying to a label in the view.

```swift
DispatchQueue.main.async {

      //Formatting each result into an array of strings
      let resultStrings = finalInferences.map({ (inference) in
        return String(format: "%s %.2f",inference.label, inference.confidence)
      })

      //Preparing them for display
      self.resultLabel.text = resultStrings.joined(separator: "\n")
}
```

App Architecture



Object Detection Model Details

MobileNet SSD trained on COCO dataset

https://github.com/tensorflow/models/tree/master/research/object_detection

COCO dataset has 90 classes

Labels file is used to list COCO classes and map to output confidences

Steps Involved in Performing Inference

1. **Initialize the Interpreter**
   Model is loaded in the interpreter at this stage
2. **Preparing the Image Input**
   Input image pixel buffer is converted to the format recognized by the model
3. **Perform Inference**
   Pass input to the Interpreter and Invoke the Interpreter
4. **Obtain and Map Results**
   Map our resulting confidence values to labels

ModelDataHandler.swift

Initializing the Interpreter

```swift
let modelPath = Bundle.main.path(forResource: modelFilename, ofType: modelFileInfo.extension)

// Specify the options for the `Interpreter`.
var options = InterpreterOptions()
options.threadCount = threadCount

interpreter = try Interpreter(modelPath: modelPath, options: options)

// Load the classes listed in the labels file.
loadLabels(fileInfo: labelsFileInfo)
```

We'll start with the interpreter initialization. As we can see, it's exactly the same as before. We load the model from the bundle, we set up the options, and we instantiate the interpreter from both. In a model like this, where the classes are in a separate labels file, we'll also load that in.

Steps Involved in Performing Inference

1 — 2 — 3 — 4

1. **Initialize the Interpreter**
   Model is loaded in the interpreter at this stage
2. **Preparing the Image Input**
   Input image pixel buffer is converted to the format recognized by the model
3. **Perform Inference**
   Pass input to the Interpreter and Invoke the Interpreter
4. **Obtain and Map Results**
   Map our resulting confidence values to labels

Step two involves preparing the image inputs. This will capture from the camera feed, getting frames in the CV pixel buffer formats. Exactly the same as before.

Preparing the Input

- Expects pixel buffer of size 300 x 300 x 3
- Our CVPixelBuffer is of type BGRA_32
- Has to be converted to Pixel buffer with only R, G, B channels.
- Pixel Buffer has to be converted to Data
- vImage is used for image operations
- https://developer.apple.com/documentation/accelerate/vimage

In the previous two examples, the mobile net model accepted input of 224 by 224. In this case, it expects 300 by 300. But other than that, the steps are exactly the same. The CV pixel buffer is in the format BGRA_32, and we need to extract the RGB channels from that, and then convert them to Data. The VM image library is used heavily for this, and you can learn more about it at this URL.

Steps Involved in Performing Inference

1 — 2 — 3 — 4

1. Initialize the Interpreter
Model is loaded in the interpreter at this stage

2. Preparing the Image Input
Input image pixel buffer is converted to the format recognized by the model

3. Perform Inference
Pass input to the Interpreter and Invoke the Interpreter

4. Obtain and Map Results
Map our resulting confidence values to labels

Invoking the Interpreter

ModelDataHandler.swift

```swift
// Copy the RGB data to the input Tensor.
try interpreter.copy(rgbData, toInputAt: 0)

// Run inference by invoking the Interpreter.
try interpreter.invoke()
```

Next up is to perform the inference. Fortunately, the code again is very similar to what you've seen already where you copy the RGB data to the input tensor and invoke the interpreter.

Where this app greatly differs from the others given that it's a different model and a totally different scenario is in the output of the model and how we obtain the class and bounding boxes.

Steps Involved in Performing Inference

1 — 2 — 3 — 4

1. Initialize the Interpreter
Model is loaded in the interpreter at this stage

2. Preparing the Image Input
Input image pixel buffer is converted to the format recognized by the model

3. Perform Inference
Pass input to the Interpreter and Invoke the Interpreter

4. Obtain and Map Results
Map our resulting confidence values to labels

Output Tensors

0	Bounding Boxes
1	Classes
2	Scores
3	Number of Results

First of all, let's consider our output tensors from the model. There's a set of four of them. The last is the number of results. So in the earlier example, we saw a banana, a mouse, and a dining table, that would be three results.

Then the bounding boxes tensor will contain the coordinates of the bounding boxes for these three as defined by the model. This will be different from the unscreened coordinates, so we'll need to do some conversion.

Next is the list of classes, which will be the identifiers for the banana, the mouse, and the dining table.

Then will be the confidence scores where we can extract the value that we saw on the screen. For example, a 90 percent chance that this object is a mouse.

ModelDataHandler.swift

Getting the Output Tensors

```
outputBoundingBox = try interpreter.output(at: 0)
outputClasses = try interpreter.output(at: 1)
outputScores = try interpreter.output(at: 2)
outputCount = try interpreter.output(at: 3)
```

The code to grab the output tensors is super simple. We simply get the interpreter output at the respective index. So the bounding boxes are at zero, the classes are at one etc.

ModelDataHandler.swift

Formatting the Results

```
// Format the results
let resultArray = formatResults(
  boundingBox: [Float](unsafeData: outputBoundingBox.data) ?? [],
  outputClasses: [Float](unsafeData: outputClasses.data) ?? [],
  outputScores: [Float](unsafeData: outputScores.data) ?? [],
  outputCount: Int(([Float](unsafeData: outputCount.data) ?? [0])[0]),
  width: CGFloat(imageWidth),
  height: CGFloat(imageHeight)
)

return resultArray
```

We can then format the contents of these tensors into a data structure to make it easy to pass around the app when we need to pass it. The bounding box, output classes, and output scores all parts neatly into float arrays. The output count is an int, so we can cast the 00 element of the array representing its tensor into an int. For convenience, we can keep the width and height of the original image. We'll use that when converting the bounding box coordinates to the [inaudible] ones so that we can draw the boxes.

ModelDataHandler.swift

Formatting the Results

```swift
func formatResults( boundingBox: [Float],
                    outputClasses: [Float],
                    outputScores: [Float],
                    outputCount: Int,
                    width: CGFloat, height: CGFloat) -> [Inference]{

var resultsArray: [Inference] = []
    for i in 0..<outputCount {
        //Obtain formatted results from the tensors
    }
}
```

The format results func can then iterate over the output count to manage the contents of each of the arrays that were mapped from the tensors, i.e. the bounding box, the output classes, and the output scores.

ModelDataHandler.swift

Formatting the Results

```swift
let score = outputScores[i]

// Filters results with confidence < threshold.
guard score >= threshold else {
    continue
}

// Gets the output class names for detected classes from labels list.
let outputClassIndex = Int(outputClasses[i])
let outputClass = labels[outputClassIndex + 1]
```

We've defined a minimum threshold for the confidence values. We filter out the results with confidence values less than this, and can then get the output class name for the results being processed by the current iteration of the loop from the label's file using the output classes tensor.

ModelDataHandler.swift

Formatting the Results

```swift
var rect: CGRect = CGRect.zero

// Translates the detected bounding box to CGRect.
rect.origin.y = CGFloat(boundingBox[4*i])
rect.origin.x = CGFloat(boundingBox[4*i+1])
rect.size.height = CGFloat(boundingBox[4*i+2]) - rect.origin.y
rect.size.width = CGFloat(boundingBox[4*i+3]) - rect.origin.x

// The detected corners are for model dimensions. So we scale the rect with respect to the
// actual image dimensions.
let newRect = rect.applying(CGAffineTransform(scaleX: width, y: height))
```

Each array in the bounding box array is an array of size 4, containing the top, left, bottom, and right corner coordinates of the detected object. I is our index variable here, so we can get our values from that multiplying by four and then offsetting. Remember that the bounding box simply contains a list of bounding box coordinates. So the values at indices zero, one, two, and three for the first bounding box. Four, five, six, and seven are for the second one etc. Thus, by multiplying by i and then offsetting we can extract the correct values.

We convert this into a CGRect. We then have to scale the obtained rect to the actual dimensions of the CV pixel buffer you pass it into the model data handler for inference. This is why we kept the original height and width of the image.

ModelDataHandler.swift

Formatting the Results

```
// Gets the color assigned for the class
let colorToAssign = colorForClass(withIndex: outputClassIndex + 1)
let inference = Inference(confidence: score,
                          className: outputClass,
                          rect: newRect,
                          displayColor: colorToAssign)
resultsArray.append(inference)
```

This code keeps it consistent, so the colors don't get randomly assigned between frames.

ViewController.swift

Formatting the Results

```
func runModel(onPixelBuffer pixelBuffer: CVPixelBuffer) {
  //Run the live camera pixelBuffer through tensorFlow to get the result
  let inferences = self.modelDataHandler?.runModel(onFrame: pixelBuffer)

  let width = CVPixelBufferGetWidth(pixelBuffer)
  let height = CVPixelBufferGetHeight(pixelBuffer)
  DispatchQueue.main.async {
  // Draws the bounding boxes and displays class names and confidence scores.
  self.drawAfterPerformingCalculations(
     onInferences: inferences,
     withImageSize: CGSize(width: CGFloat(width), height: CGFloat(height)))
  }
}
```

So then if we return to the view controller, which runs the model, e can see that after it gets the inferences, it calls a draw after performing calculations function passing in the inferences. Remember that these inferences now contain the bounding box coordinates, so becomes an easy task to draw them out.