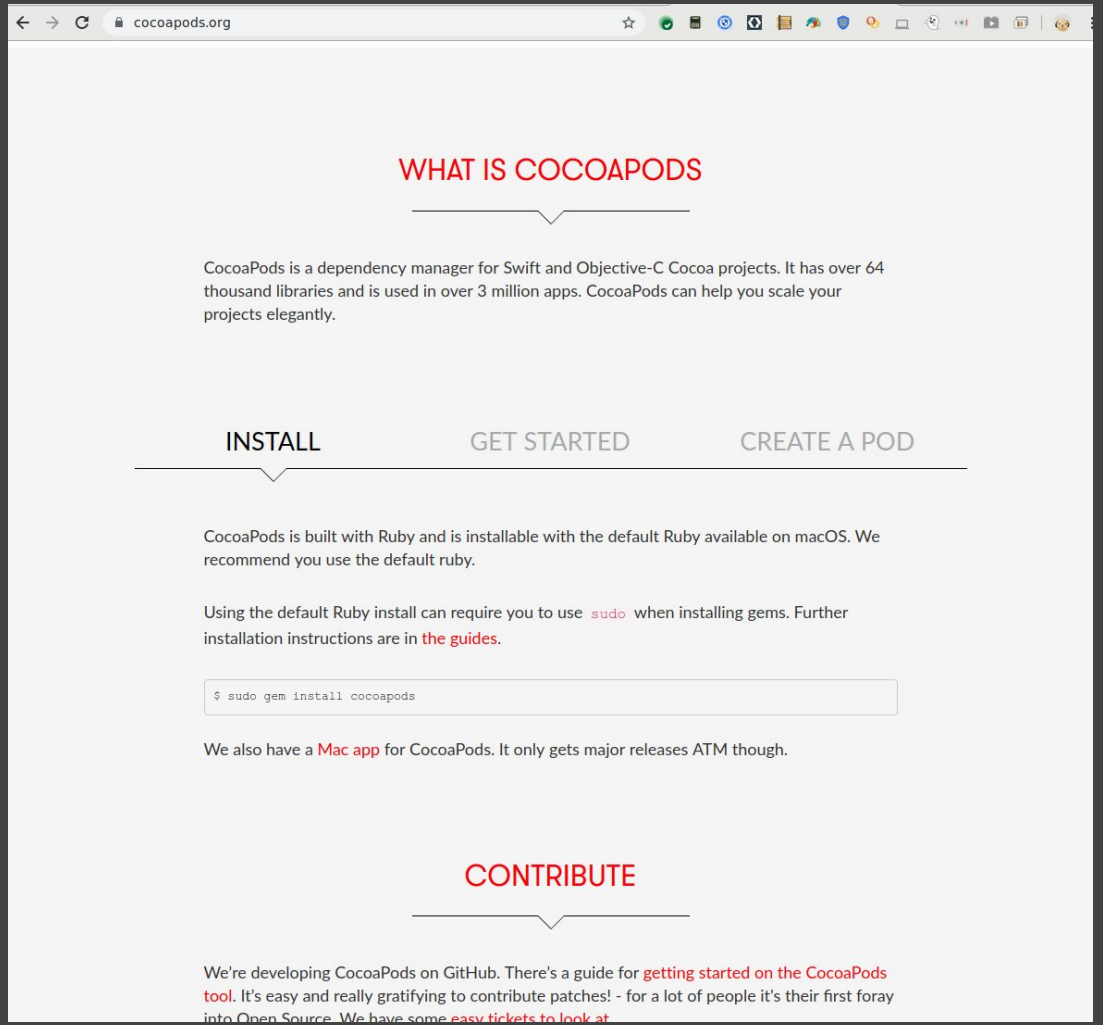
Copyright Notice

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

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



Podfile

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

Install Command

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

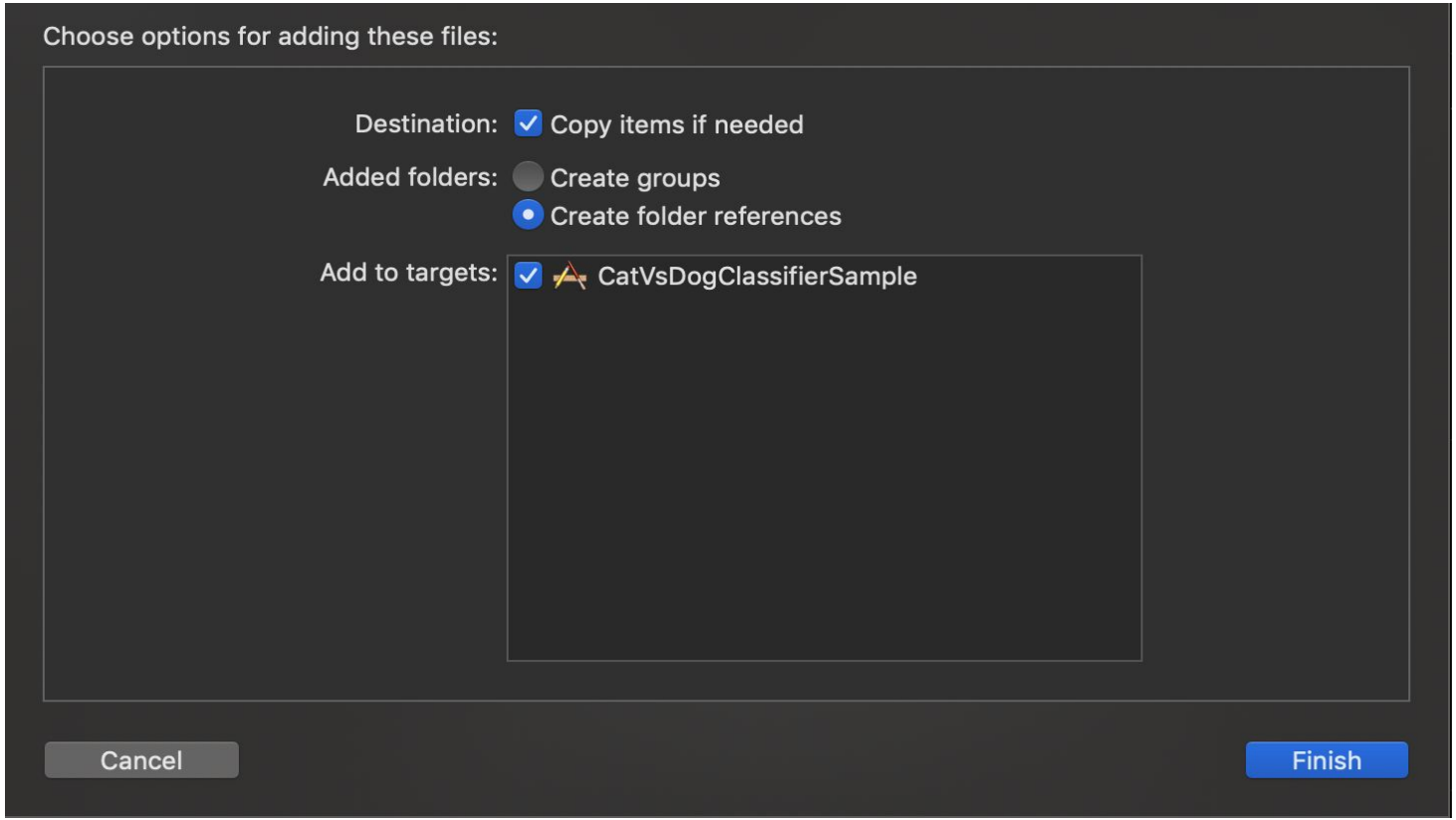
Getting the Model

- Python notebook to train the model:

  bit.ly/makecatsdogs

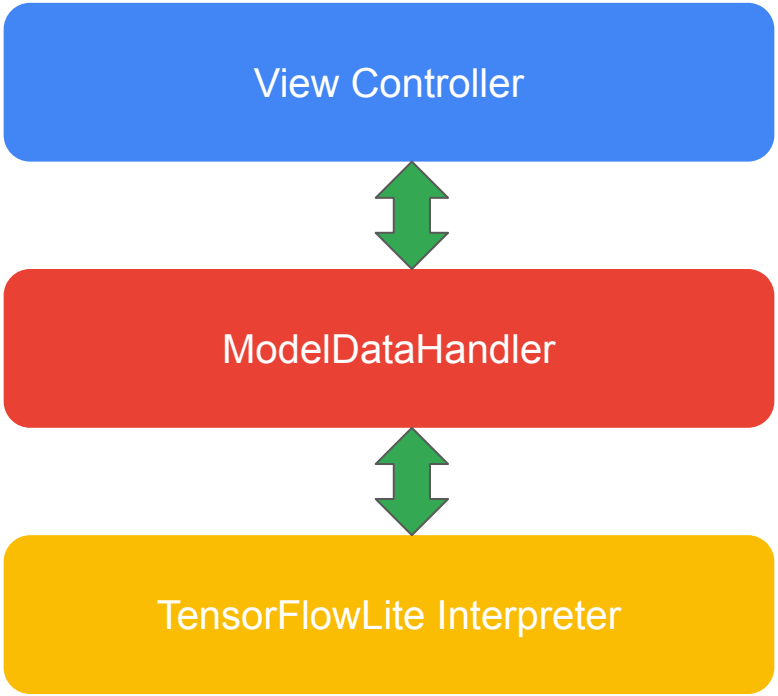
- After running all the cells, you get model(.tflite) and labels(.txt) files

Adding Model and Labels




Carrier
9:10 PM
Please click on the images to perform inference
Dog

App Architecture



Interpreter

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

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

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

```
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

```
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

```
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

```
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

```
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 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… CVPixelBuffer and normalize
- Final output h… 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

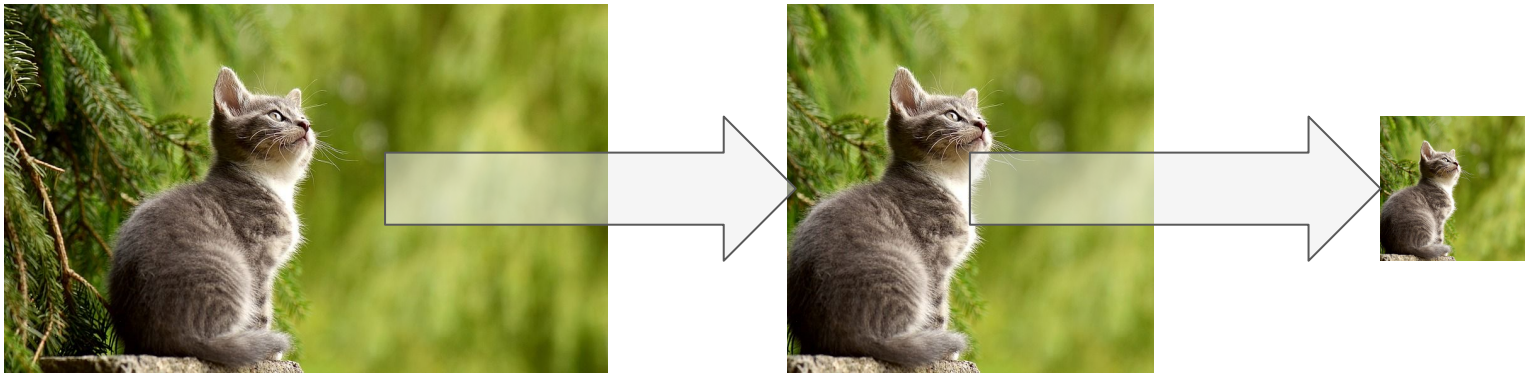
- Model Expects pixel buffer of size 224 x 224 x 3
- iOS uses CVPixelBuffer to represent images in memory
- CVPixelBuffer has Alpha as well as RGB
- Need to extract R, G, B from CVPixelBuffer and normalize
- Final output has to be of type ‘Data’

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’

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

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

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

ModelDataHandler.swift

```
let inputChannels = 3

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

ModelDataHandler.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)


}
```

ModelDataHandler.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
}
```

ModelDataHandler.swift

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

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

```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)
```

ModelDataHandler.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)
```

ModelDataHandler.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)
```

ModelDataHandler.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

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

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

ModelDataHandler.swift

```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]) }
 }
```

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]) }
 }
```

ModelDataHandler.swift

```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]) }
 }
```

ModelDataHandler.swift

```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]) }
 }
```

Calling Inference from UI

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

ViewController.swift

Initializing the ModelDataHandler

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

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)

}
```

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)

}
```

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)

}
```

ViewController.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)

}
```

ViewController.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)

}
```

8:08

Please click on the images to perform inference

Inferred label

Dog

App Architecture

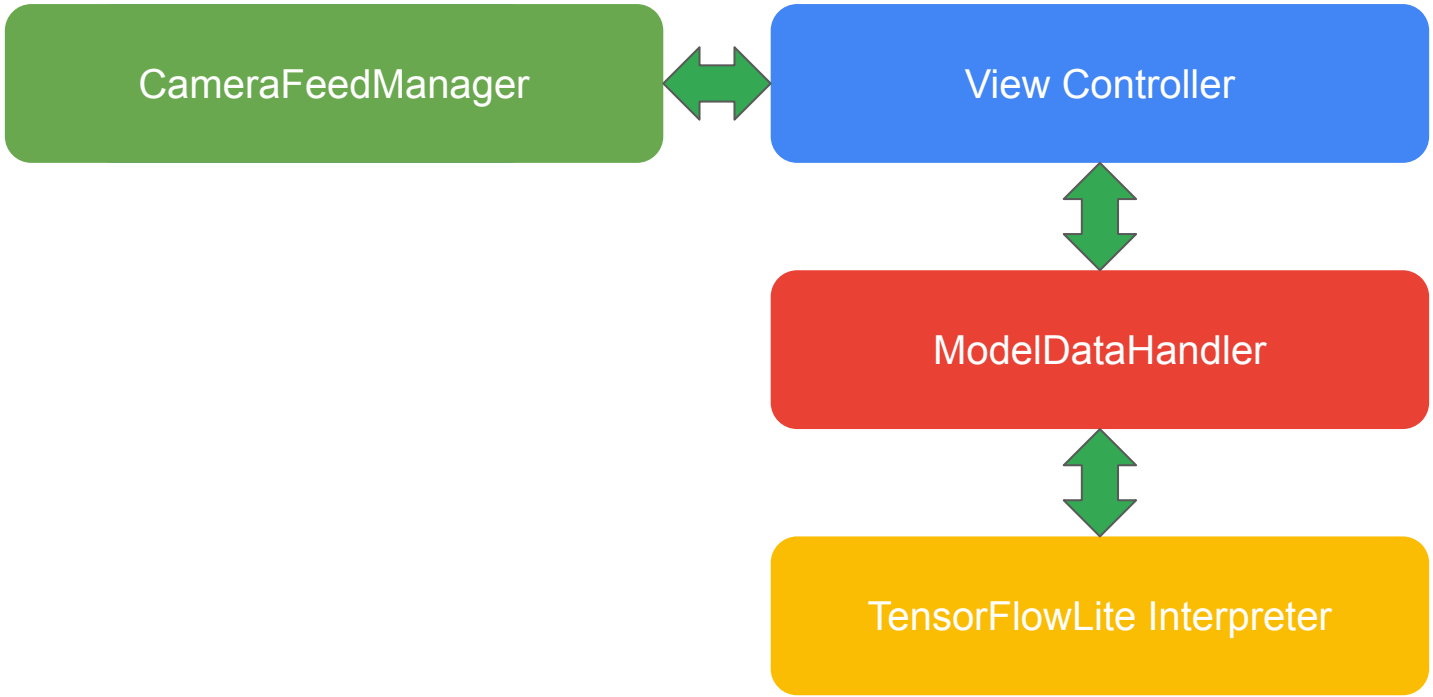


Image Classification Model Details

- 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

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

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

```
  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)
```

ModelDataHandler.swift

```
do {
    // Allocate memory for the model's input `Tensor`s.
    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

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

ViewController.swift

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

override func viewWillAppear(_ animated: Bool) {

    cameraCapture.checkCameraConfigurationAndStartSession()

}

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

ViewController.swift

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

override func viewWillAppear(_ animated: Bool) {

    cameraCapture.checkCameraConfigurationAndStartSession()

}

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

ViewController.swift

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

override func viewWillAppear(_ animated: Bool) {

    cameraCapture.checkCameraConfigurationAndStartSession()

}

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

ViewController.swift

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

override func viewWillAppear(_ animated: Bool) {

    cameraCapture.checkCameraConfigurationAndStartSession()

}

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

ViewController.swift

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

CameraFeedManager.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)
    }
}
```

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 })
```

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

Invoking the Interpreter

```
// 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

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 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

```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]) }
 }
```

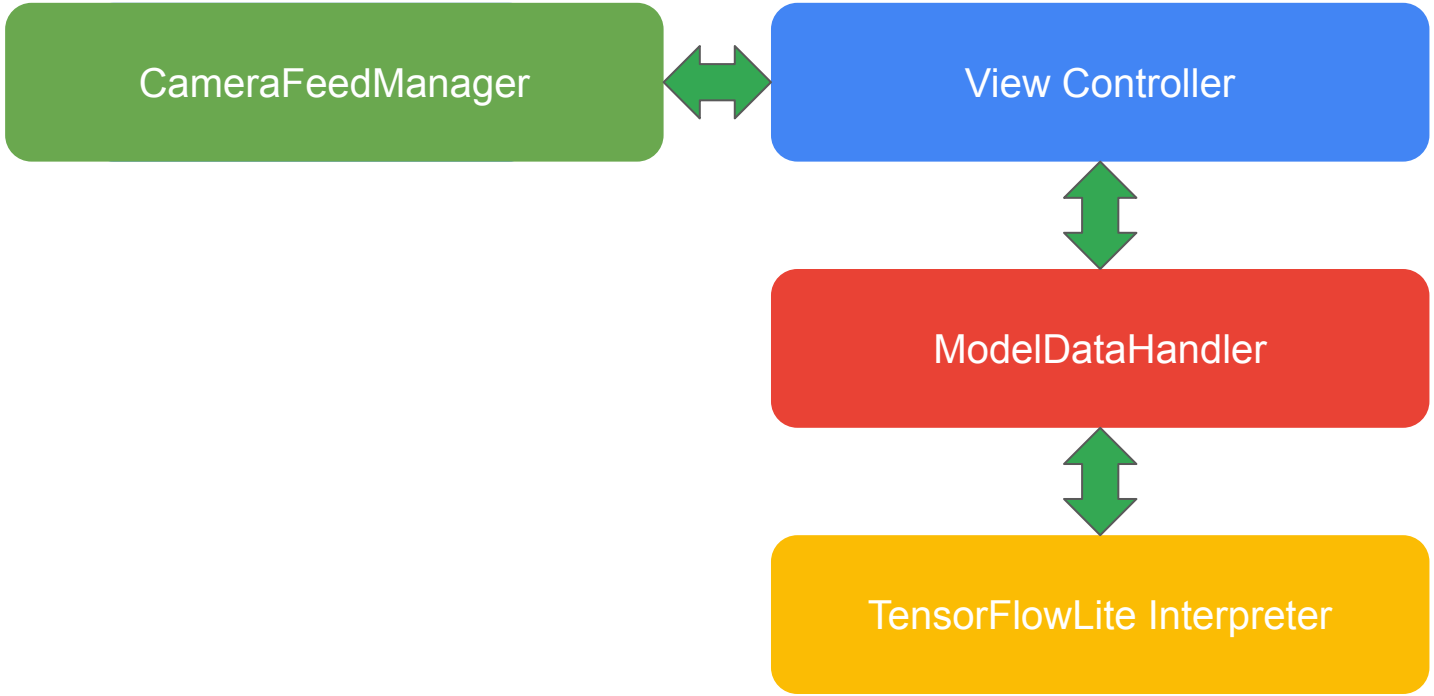
ViewController.swift

```
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

```
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)
```

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

- 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

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

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()
```

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

Output Tensors

0	Bounding Boxes
1	Classes
2	Scores
3	Number of Results

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)
```

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
```

ModelDataHandler.swift

Formatting the Results

```
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
    }
}
```

ModelDataHandler.swift

Formatting the Results

```
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]
```

ModelDataHandler.swift

Formatting the Results

```
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))
```

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)
```

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)))
  }
}
```

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)))
  }
}
```