Sample Code

Detecting human body poses in an image

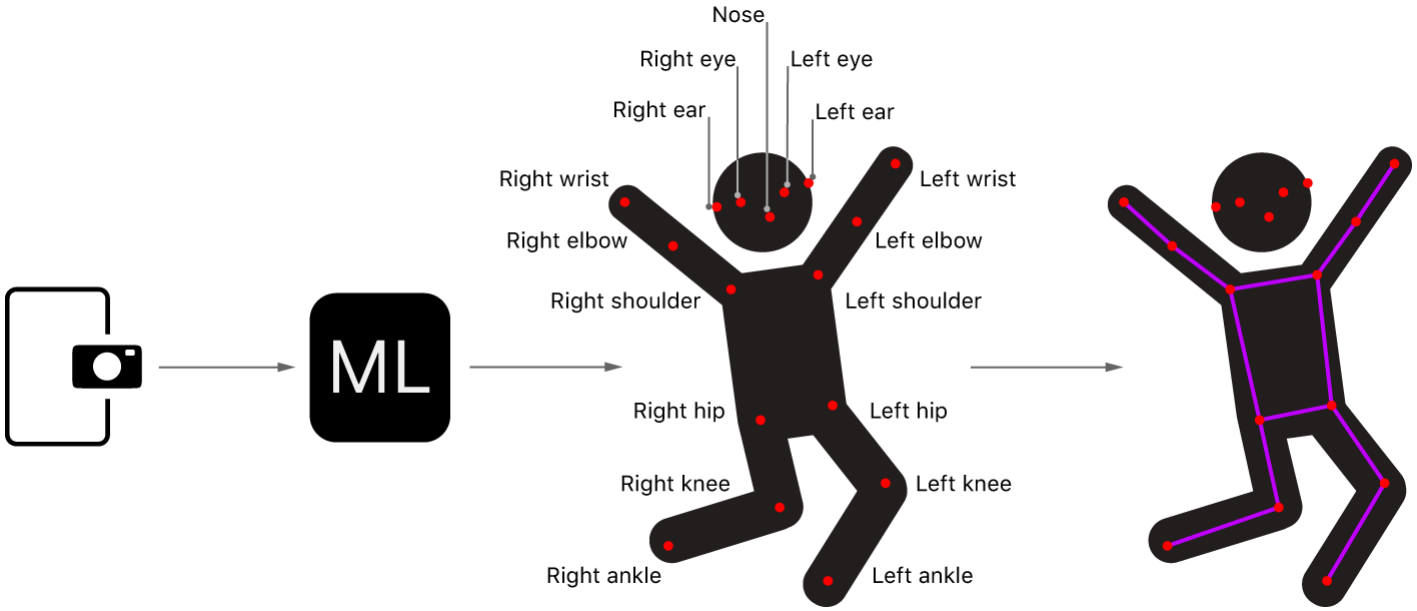
Locate people and the stance of their bodies by analyzing an image with a PoseNet model.

Download

iOS 13.0+ | iPadOS 13.0+ | Xcode 15.2+

Overview

This sample project provides an illustrative example of using a third-party Core ML model, PoseNet, to detect human body poses from frames captured using a camera. PoseNet models detect 17 different body parts or joints: eyes, ears, nose, shoulders, hips, elbows, knees, wrists, and ankles. Collectively these joints form a pose.



The sample finds the locations of the 17 joints for each person in the image and draws a wireframe pose on top of them.

> **Note**
>
> Starting in iOS 14 and macOS 11, Vision adds the ability to detect human body poses. For details, see Detecting Human Body Poses in Images.

Configure the capture session

The sample starts by getting an image from the device's built-in camera using an `AVCaptureSession` (see Setting Up a Capture Session).

```
if captureSession.isRunning {
    captureSession.stopRunning()
}

captureSession.beginConfiguration()

captureSession.sessionPreset = .vga640x480

try setCaptureSessionInput()

try setCaptureSessionOutput()

captureSession.commitConfiguration()
```

Acquire the captured image

```
// Attempt to lock the image buffer to gain access to its memory.
guard CVPixelBufferLockBaseAddress(pixelBuffer, .readOnly) == kCVReturnSuccess
    else {
        return
}

// Create Core Graphics image placeholder.
var image: CGImage?

// Create a Core Graphics bitmap image from the pixel buffer.
VTCreateCGImageFromCVPixelBuffer(pixelBuffer, options: nil, imageOut: &image)

// Release the image buffer.
```

```
CVPixelBufferUnlockBaseAddress(pixelBuffer, .readOnly)

DispatchQueue.main.sync {
    delegate.videoCapture(self, didCaptureFrame: image)
}
```

Prepare the input for the PoseNet model

```
// Wrap the image in an instance of PoseNetInput to have it resized
// before being passed to the PoseNet model.
let input = PoseNetInput(image: image, size: self.modelInputSize)
```

Pass the input to the PoseNet model

The sample app then proceeds to pass the input to the PoseNet's `prediction` function to obtain its outputs, which the app uses to detect poses.

```
guard let prediction = try? self.poseNetMLModel.prediction(from: input) else {
    return
}
```

```
let poseNetOutput = PoseNetOutput(prediction: prediction,
                                  modelInputSize: self.modelInputSize,
                                  modelOutputStride: self.outputStride)

DispatchQueue.main.async {
    self.delegate?.poseNet(self, didPredict: poseNetOutput)
}
```

Analyze the PoseNet output to locate joints

The sample uses one of two algorithms to locate the joints of either one person or multiple persons. The single-person algorithm, the simplest and fastest, inspects the model's outputs to locate the most prominent joints in the image and uses these joints to construct a single pose.

```
var pose = Pose()

// For each joint, find its most likely position and associated confidence
// by querying the heatmap array for the cell with the greatest
```

```
// confidence and using this to compute its position.
pose.joints.values.forEach { joint in
    configure(joint: joint)
}

// Compute and assign the confidence for the pose.
pose.confidence = pose.joints.values
    .map { $0.confidence }.reduce(0, +) / Double(Joint.numberOfJoints)

// Map the pose joints positions back onto the original image.
pose.joints.values.forEach { joint in
    joint.position = joint.position.applying(modelToInputTransformation)
}

return pose
```

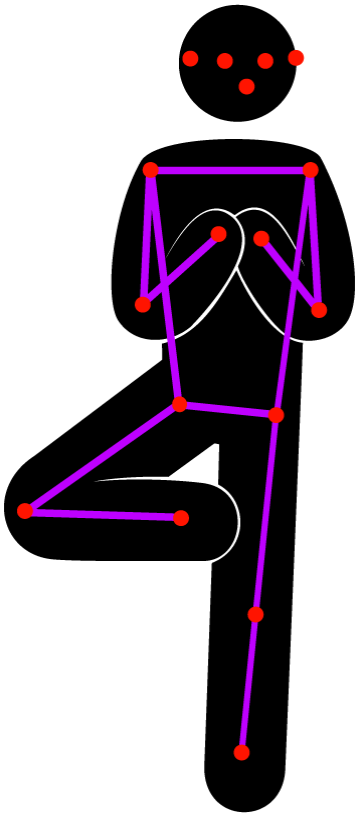
The multiple-person algorithm first identifies a set of candidate root joints as starting points. It uses these root joints to find neighboring joints and repeats the process until it has located all 17 joints of each person. For example, the algorithm may find a left knee with a high confidence, and then search for its adjacent joints, the left ankle and left hip.

```
var detectedPoses = [Pose]()

// Iterate through the joints with the greatest confidence, referred to here as
// candidate roots, using each as a starting point to assemble a pose.
for candidateRoot in candidateRoots {
    // Ignore any candidates that are in the proximity of joints of the
    // same type and have already been assigned to an existing pose.
    let maxDistance = configuration.matchingJointDistance
    guard !detectedPoses.contains(candidateRoot, within: maxDistance) else {
        continue
    }

    var pose = assemblePose(from: candidateRoot)

    // Compute the pose's confidence by dividing the sum of all
    // non-overlapping joints, from existing poses, by the total
    // number of joints.
    pose.confidence = confidence(for: pose, detectedPoses: detectedPoses)

    // Ignore any pose that has a confidence less than the assigned threshold.
    guard pose.confidence >= configuration.poseConfidenceThreshold else {
```

```
            continue
        }

        detectedPoses.append(pose)

        // Exit early if enough poses have been detected.
        if detectedPoses.count >= configuration.maxPoseCount {
            break
        }
    }

    // Map the pose joints positions back onto the original image using
    // the pre-computed transformation matrix.
    detectedPoses.forEach { pose in
        pose.joints.values.forEach { joint in
            joint.position = joint.position.applying(modelToInputTransformation)
        }
    }

    return detectedPoses
```

Visualize the detected poses

For each detected pose, the sample app draws a wireframe over the input image, connecting the lines between the joints and then drawing circles for the joints themselves.



```
let dstImageSize = CGSize(width: frame.width, height: frame.height)
let dstImageFormat = UIGraphicsImageRendererFormat()

dstImageFormat.scale = 1
let renderer = UIGraphicsImageRenderer(size: dstImageSize,
                                       format: dstImageFormat)

let dstImage = renderer.image { rendererContext in
    // Draw the current frame as the background for the new image.
    draw(image: frame, in: rendererContext.cgContext)

    for pose in poses {
        // Draw the segment lines.
        for segment in PoseImageView.jointSegments {
            let jointA = pose[segment.jointA]
            let jointB = pose[segment.jointB]

            guard jointA.isValid, jointB.isValid else {
                continue
            }

            drawLine(from: jointA,
                     to: jointB,
                     in: rendererContext.cgContext)
        }

        // Draw the joints as circles above the segment lines.
        for joint in pose.joints.values.filter({ $0.isValid }) {
            draw(circle: joint, in: rendererContext.cgContext)
        }
    }
}
```

See Also

Image classification models

{} Using Core ML for semantic image segmentation

Identify multiple objects in an image by using the DEtection TRansformer image-segmentation model.

{} Classifying Images with Vision and Core ML

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

{} Understanding a Dice Roll with Vision and Object Detection

Detect dice position and values shown in a camera frame, and determine the end of a roll by leveraging a dice detection model.