Sample Code

Detecting human actions in a live video feed

Identify body movements by sending a person's pose data from a series of video frames to an action-classification model.

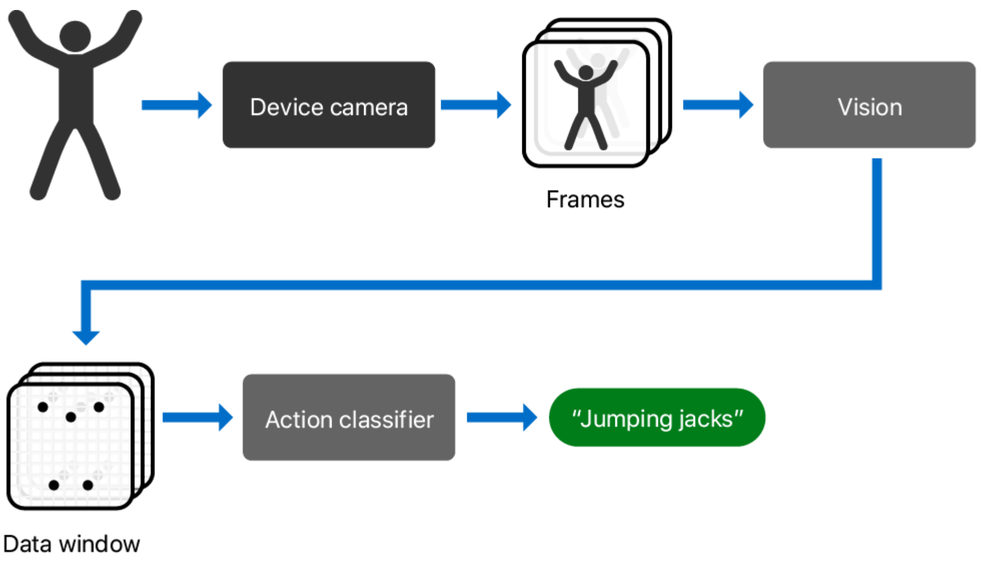
Download

iOS 14.0+ | iPadOS 14.0+ | Xcode 12.3+

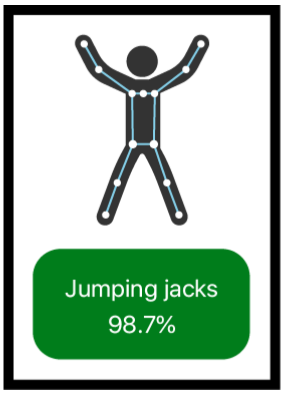
Overview

This sample app recognizes a person's body moves, called *actions*, by analyzing a series of video frames with Vision and predicting the name of the movement by applying an action classifier. The action classifier in this sample recognizes three exercises:

- Jumping jacks
- Lunges
- Burpees



The app continually presents its current action prediction on top of a live, full-screen video feed from the device's camera. When the app recognizes one or more people in the frame, it overlays a wireframe body pose on each person. At the same time, the app predicts the *prominent* person's current action; typically this is the person closest to the camera.



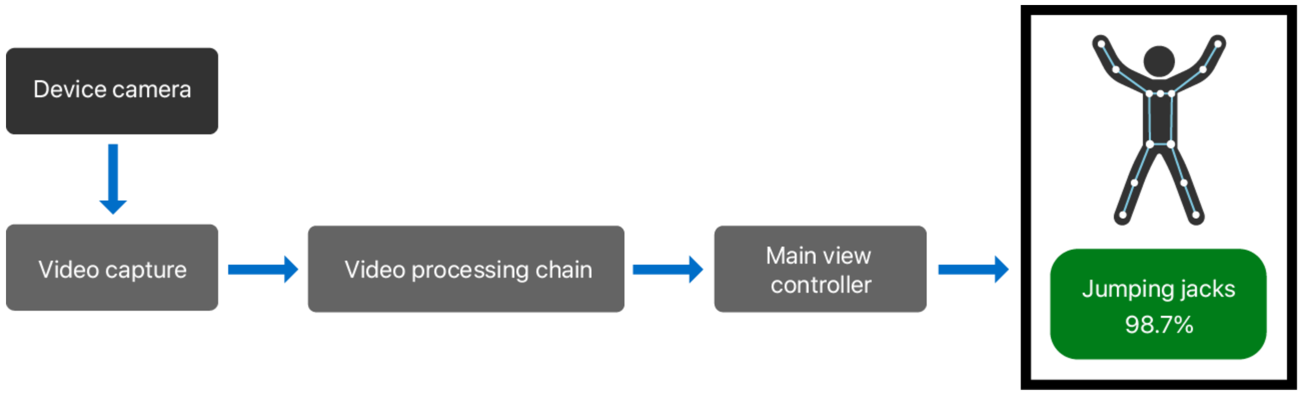
At launch, the app configures the device's camera to generate video frames and then directs the frames through a series of methods it chains together with Combine. These methods work together to analyze the frames and make action predictions by performing the following sequence of steps:

1. Locate all human body poses in each frame.

2. Isolate the prominent pose.

3. Aggregate the prominent pose's position data over time.

4. Make action predictions by sending the aggregate data to the action classifier.



Configure the Sample Code Project

This sample app uses a camera, so you can't run it in Simulator — you need to run it on an iOS or iPadOS device.

Start a Video Capture Session

The app’s `VideoCapture`class configures the device’s camera to generate video frames by creating an `AVCaptureSession`.

When the app first launches, or when the user rotates the device or switches between cameras, video capture configures a camera input, a frame output, and the connection between them in its `configureCaptureSession()` method.

```
let input = AVCaptureDeviceInput.createCameraInput(position: cameraPosition)

let output = AVCaptureVideoDataOutput.withPixelFormatType(kCVPixelFormatType_32BGRA)

let success = configureCaptureConnection(input, output)
return success ? output : nil
```

The `createCameraInput(position:frameRate:)` method selects the front- or rear-facing camera and configures its frame rate so it matches that of the action classifier.

Important

If you replace the `ExerciseClassifier.mlmodel` file with your own action classifier model, set the `frameRate` property to match the Frame Rate training parameter you used in the Create ML developer tool.

The `AVCaptureVideoDataOutput.withPixelFormatType(_:)` method creates an `AVCaptureVideoDataOutput` that produces frames with a specific pixel format.

The `configureCaptureConnection(_:_:)` method configures the relationship between the capture session’s camera input and video output by:

- Selecting a video orientation
- Deciding whether to horizontally flip the video
- Enabling image stabilization when applicable

```
if connection.isVideoOrientationSupported {
    // Set the video capture's orientation to match that of the device.
    connection.videoOrientation = orientation
}

if connection.isVideoMirroringSupported {
    connection.isVideoMirrored = horizontalFlip
}
```

```
if connection.isVideoStabilizationSupported {
    if videoStabilizationEnabled {
        connection.preferredVideoStabilizationMode = .standard
    } else {
        connection.preferredVideoStabilizationMode = .off
    }
}
```

The method keeps the app operating in real time — and avoids building up a frame backlog — by setting the video output's `alwaysDiscardsLateVideoFrames` property to `true`.

```
// Discard newer frames if the app is busy with an earlier frame.
output.alwaysDiscardsLateVideoFrames = true
```

See Setting up a capture session for more information on how to configure capture sessions and connect their inputs and outputs.

Create a Frame Publisher

The video capture publishes frames from its capture session by creating a `PassthroughSubject` in its `createVideoFramePublisher()` method.

```
// Create a new passthrough subject that publishes frames to subscribers.
let passthroughSubject = PassthroughSubject<Frame, Never>()

// Keep a reference to the publisher.
framePublisher = passthroughSubject
```

A passthrough subject is a concrete implementation of `Subject` that adapts imperative code to work with Combine. It immediately publishes the instance you pass to its `send(_:)` method, if it has a subscriber at that time.

Next, the video capture registers itself as the video output's delegate so it receives the video frames from the capture session by calling the output's `setSampleBufferDelegate(_:queue:)` method.

```
// Set the video capture as the video output's delegate.
videoDataOutput.setSampleBufferDelegate(self, queue: videoCaptureQueue)
```

The video capture forwards each frame it receives to its `framePublisher` by passing the frame to its `send(_:)` method.

```
extension VideoCapture: AVCaptureVideoDataOutputSampleBufferDelegate {
    func captureOutput(_ output: AVCaptureOutput,
                       didOutput frame: Frame,
                       from connection: AVCaptureConnection) {

        // Forward the frame through the publisher.
        framePublisher?.send(frame)
    }
}
```

Build a Publisher Chain

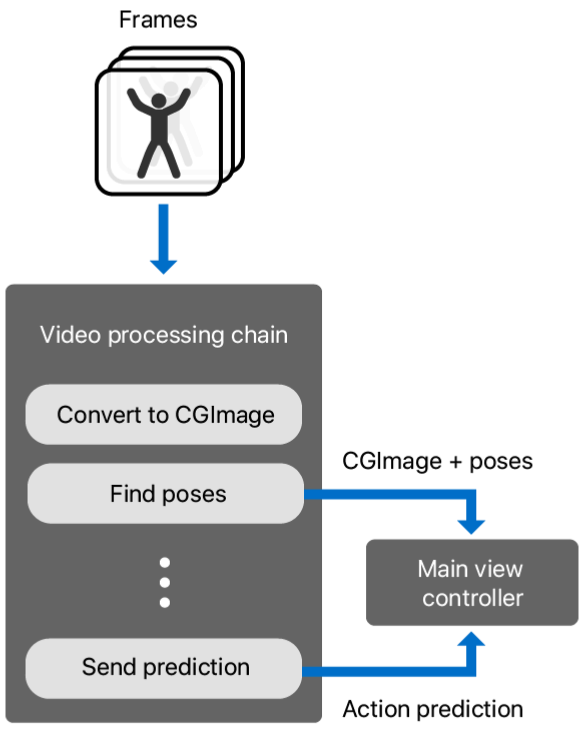
The sample processes each video frame, and its derivative data, with a series of methods that it connects together into a chain of Combine publishers in the `VideoProcessingChain` class.

Each time the video capture creates a new frame publisher it notifies the main view controller, which then assigns the publisher to the video-processing chain's `upstreamFramePublisher` property:

```
func videoCapture(_ videoCapture: VideoCapture,
                  didCreate framePublisher: FramePublisher) {
    updateUILabelsWithPrediction(.startingPrediction)

    // Build a new video-processing chain by assigning the new frame publisher.
    videoProcessingChain.upstreamFramePublisher = framePublisher
}
```

Each time the property's value changes, the video-processing chain creates a new daisy chain of publishers by calling its `buildProcessingChain()` method.



The method creates each new publisher by calling one of the following `Publisher` methods:

- `map(_:)`
- `compactMap(_:)`
- `scan(_:_:)`
- `filter(_:)`

For example, the publisher that subscribes to the initial frame publisher is a `Publishers.CompactMap` that converts each `Frame` (a type alias of `CMSampleBuffer`) it receives into a `CGImage` by calling the video-processing chain's `imageFromFrame(_:)` method.

```
// Create the chain of publisher-subscribers that transform the raw video
// frames from upstreamFramePublisher.
frameProcessingChain = upstreamFramePublisher
    // ---- Frame (aka CMSampleBuffer) -- Frame ----

    // Convert each frame to a CGImage, skipping any that don't convert.
    .compactMap(imageFromFrame)

    // ---- CGImage -- CGImage ----

    // Detect any human body poses (or lack of them) in the frame.
    .map(findPosesInFrame)

    // ---- [Pose]? -- [Pose]? ----
```

The next sections explain the remaining publishers in the chain and the methods they use to transform their inputs.

Analyze Each Frame for Body Poses

The next publisher in the chain is a `Publishers.Map` that receives each `CGImage` from the previous publisher (the compact map) by subscribing to it. The map publisher locates any human body poses in the frame by using the video-processing chain's `findPosesInFrame(_:)` method. The method invokes a `VNDetectHumanBodyPoseRequest` by creating a `VNImageRequestHandler` with the image and submitting the video-processing chain's `humanBodyPoseRequest` property to the handler's `perform(_:)` method.

Important

Improve your app's efficiency by creating and reusing a single `VNDetectHumanBodyPoseRequest` instance.

```
// Create a request handler for the image.
let visionRequestHandler = VNImageRequestHandler(cgImage: frame)

// Use Vision to find human body poses in the frame.
do { try visionRequestHandler.perform([humanBodyPoseRequest]) } catch {
    assertionFailure("Human Pose Request failed: \(error)")
}
```

When the request completes, the method creates and returns a `Pose` array that contains one pose for every `VNHumanBodyPoseObservation` instance in the request's `results` property.

```
let poses = Pose.fromObservations(humanBodyPoseRequest.results)
```

The `Pose` structure in this sample serves three main purposes:

- Calculating the observation's area within a frame (see "Isolate A Body Pose")
- Storing the the observation's multiarray (see "Retrieve the Multiarray")
- Drawing an observation as a wireframe of points and lines (see "Present the Poses to the User")

For more information about using a `VNDetectHumanBodyPoseRequest`, see Detecting Human Body Poses in Images.

Isolate a Body Pose

The next publisher in the chain is a map that chooses a single pose from the array of poses by using the video-processing chain's `isolateLargestPose(_:)` method. This method selects the the most prominent pose by passing a closure to the pose array's `max(by:)` method.

```
private func isolateLargestPose(_ poses: [Pose]?) -> Pose? {
    return poses?.max(by:) { pose1, pose2 in pose1.area < pose2.area }
}
```

The closure compares the poses' area estimates, with the goal of consistently selecting the same person's pose over time, when multiple people are in frame.

Important

Get the most accurate predictions from an action classifier by using whatever technique you think best tracks a person from frame to frame, and use the multiarray from that person's `VNHumanBodyPoseObservation` result.

Retrieve the Multiarray

The next publisher in the chain is a map that publishes the `MLMultiArray` from the pose's `multiArray` property by using the video processing chain's `multiArrayFromPose(_:)` method.

```
private func multiArrayFromPose(_ item: Pose?) -> MLMultiArray? {
    return item?.multiArray
}
```

The `Pose` initializer copies the multiarray from its `VNHumanBodyPoseObservation` parameter by calling the observation's `keypointsMultiArray()` method.

```
// Save the multiarray from the observation.
multiArray = try? observation.keypointsMultiArray()
```

Gather a Window of Multiarrays

The next publisher in the chain is a `Publishers.Scan` that receives each multiarray from its upstream publisher and gathers them into an array by providing two arguments:

- An empty multiarray-optional array as the scan publisher's initial value.
- The video-processing chain's `gatherWindow(previousWindow:multiArray:)` method as the scan publisher's transform.

```
// ---- MLMultiArray? -- MLMultiArray? ----

// Gather a window of multiarrays, starting with an empty window.
.scan([MLMultiArray?](), gatherWindow)

// ---- [MLMultiArray?] -- [MLMultiArray?] ----
```

A scan publisher behaves similarly to a map, but it also maintains a state. The following scan publisher's state is an array of multiarray optionals that's initially empty. As the scan publisher receives multiarray optionals from its upstream publisher, the scan publisher passes its previous state and the incoming multiarray optional as arguments to its transform.

```
private func gatherWindow(previousWindow: [MLMultiArray?],
                          multiArray: MLMultiArray?) -> [MLMultiArray?] {
    var currentWindow = previousWindow

    // If the previous window size is the target size, it
    // means sendWindowWhenReady() just published an array window.
    if previousWindow.count == predictionWindowSize {
        // Advance the sliding array window by stride elements.
        currentWindow.removeFirst(windowStride)
    }

    // Add the newest multiarray to the window.
    currentWindow.append(multiArray)

    // Publish the array window to the next subscriber.
    // The currentWindow becomes this method's next previousWindow when
    // it receives the next multiarray from the upstream publisher.
    return currentWindow
}
```

The method:

1. Copies the `previousWindow` parameter to `currentWindow`
2. Removes `windowStride` elements from the front of `currentWindow`, if it's full

3. Appends the `multiArray` parameter to the end of `currentWindow`
4. Returns `currentWindow`, which becomes the new state of the scan publisher and the next value for `previousWindow` when the scan publisher receives the next value from its upstream publisher and invokes the method

The video-processing chain considers a window to be full if it contains `predictionWindowSize` elements. When the window is full, this method removes (in step 2) the oldest elements to make room for newer elements, effectively sliding the window forward in time.

The Exercise Classifier's `calculatePredictionWindowSize()` method determines the value of the prediction window size at runtime by inspecting the model's `modelDescription` property.

Monitor the Window Size

The next publisher in the chain is a `Publishers.Filter`, which only publishes an array window when the `gateWindow(_:)` method returns `true`.

```
// Only publish a window when it grows to the correct size.
.filter(gateWindow)

// ---- [MLMultiArray?] -- [MLMultiArray?] ----
```

The method returns `true` if the window array contains exactly the number of elements defined in `predictionWindowSize`. Otherwise, the method returns `false`, which instructs the filter publisher to discard the current window and not publish it.

```
private func gateWindow(_ currentWindow: [MLMultiArray?]) -> Bool {
    return currentWindow.count == predictionWindowSize
}
```

This filter publisher, in combination with its upstream scan publisher, publishes an array of multiarray optionals once per each number of frames defined in `windowStride`.

Predict the Person's Action

The next publisher in the chain makes an `ActionPrediction` from the multiarray window by using the `predictActionWithWindow(_:)` method as its transform.

```
// Make an activity prediction from the window.
.map(predictActionWithWindow)

// ---- ActionPrediction -- ActionPrediction ----
```

The method’s input array contains multiarray optionals where each `nil` element represents a frame in which Vision wasn’t able to find any human body poses. An action classifier requires a valid, non-`nil` multiarray for every frame. To remove the `nil` elements in the array, the method creates a new multiarray, `filledWindow`, by:

- Copying each each valid element in `currentWindow`
- Replacing each `nil` element in `currentWindow` with an `emptyPoseMultiArray`

```
var poseCount = 0

// Fill the nil elements with an empty pose array.
let filledWindow: [MLMultiArray] = currentWindow.map { multiArray in
    if let multiArray = multiArray {
        poseCount += 1
        return multiArray
    } else {
        return Pose.emptyPoseMultiArray
    }
}
```

The empty pose multiarray has:

- Every element set to zero
- The same value for its shape property as a multiarray from a human body-pose observation

As the method iterates through each element in `currentWindow`, it tallies the number of non-`nil` elements with `poseCount`.

If the value of `poseCount` is too low, the method directly creates a `noPersonPrediction` action prediction.

```
// Only use windows with at least 60% real data to make a prediction
// with the action classifier.
let minimum = predictionWindowSize * 60 / 100
guard poseCount >= minimum else {
    return ActionPrediction.noPersonPrediction
}
```

Otherwise, the method merges the array of multiarrays into a single, combined multiarray by calling the `init(byConcatenatingMultiArrays:alongAxis:dataType:)` initializer.

```
// Merge the array window of multiarrays into one multiarray.
let mergedWindow = MLMultiArray(concatenating: filledWindow,
                                axis: 0,
                                dataType: .float)
```

The method generates an action prediction by passing the combined multiarray to the action classifier's `predictActionFromWindow(_:)` helper method.

```
// Make a genuine prediction with the action classifier.
let prediction = actionClassifier.predictActionFromWindow(mergedWindow)

// Return the model's prediction if the confidence is high enough.
// Otherwise, return a "Low Confidence" prediction.
return checkConfidence(prediction)
```

The method checks the prediction's confidence by passing the prediction to the `checkConfidence(_:)` helper method, which returns the same prediction if its confidence is high enough; otherwise `lowConfidencePrediction`.

Present the Prediction to the User

The final component in the chain is a subscriber that notifies the video-processing chain's delegate with the prediction using the `sendPrediction(_:)` method.

```
// Send the action prediction to the delegate.
.sink(receiveValue: sendPrediction)
```

The method sends the action prediction and the number of frames the prediction represents (`windowStride`) to the video-processing chain's `delegate`, the main view controller.

```
// Send the prediction to the delegate on the main queue.
DispatchQueue.main.async {
    self.delegate?.videoProcessingChain(self,
                                        didPredict: actionPrediction,
                                        for: windowStride)
}
```

Each time the main view controller receives an action prediction, it updates the app's UI with the prediction and confidence in a helper method.

```
func videoProcessingChain(_ chain: VideoProcessingChain,
                          didPredict actionPrediction: ActionPrediction,
                          for frameCount: Int) {

    if actionPrediction.isModelLabel {
        // Update the total number of frames for this action.
        addFrameCount(frameCount, to: actionPrediction.label)
    }

    // Present the prediction in the UI.
    updateUILabelsWithPrediction(actionPrediction)
}
```

The main view controller also updates its `actionFrameCounts` property for action labels that come from the model, which it later sends to the Summary View Controller when the user taps the `Summary` button.

Present the Poses to the User

The app visualizes the result of each human body-pose request by drawing the poses on top of the frame in which Vision found them. Each time the video-processing chain's `findPosesInFrame(_:)` creates an array of `Pose` instances, it sends the poses to its delegate, the main view controller.

```
// Send the frame and poses, if any, to the delegate on the main queue.
DispatchQueue.main.async {
    self.delegate?.videoProcessingChain(self, didDetect: poses, in: frame)
}
```

The main view controller's `drawPoses(_:onto:)` method uses the frame as the background by first drawing the frame.

```
// Draw the camera image first as the background.
let imageRectangle = CGRect(origin: .zero, size: frameSize)
cgContext.draw(frame, in: imageRectangle)
```

Next, the method draws the poses by calling their `drawWireframeToContext(_:applying:)` method, which draws the pose as a wireframe of lines and circles.

```
// Draw all the poses Vision found in the frame.
for pose in poses {
    // Draw each pose as a wireframe at the scale of the image.
    pose.drawWireframeToContext(cgContext, applying: pointTransform)
}
```

The main view controller presents the finished image to the user by assigning it to its full-screen image view.

```
// Update the UI's full-screen image view on the main thread.
DispatchQueue.main.async { self.imageView.image = frameWithPosesRendering }
```