Pose landmark detection guide for Android

Send feedback

The MediaPipe Pose Landmarker task lets you detect landmarks of human bodies in an image or video. You can use this task to identify key body locations, analyze posture, and categorize movements. This task uses machine learning (ML) models that work with single images or video. The task outputs body pose landmarks in image coordinates and in 3-dimensional world coordinates.

The code sample described in these instructions is available on GitHub. For more information about the capabilities, models, and configuration options of this task, see the Overview.

The MediaPipe Tasks example code is a simple implementation of a Pose Landmarker app for Android. The example uses the camera on a physical Android device to detect poses in a continuous video stream. The app can also detect poses in images and videos from the device gallery.

You can use the app as a starting point for your own Android app, or refer to it when modifying an existing app. The

Pose Landmarker example code is hosted on GitHub.

Download the code

The following instructions show you how to create a local copy of the example code using the git command line tool.

Attention: This MediaPipe Solutions Preview is an early release. Learn more.

To download the example code:

1. Clone the git repository using the following command:

   ```
   git clone https://github.com/google-ai-edge/mediapipe-samples
   ```

2. Optionally, configure your git instance to use sparse checkout, so you have only the files for the Pose Landmarker example app:

   ```
   cd mediapipe-samples
   git sparse-checkout init --cone
   git sparse-checkout set examples/pose_landmarker/android
   ```

After creating a local version of the example code, you can import the project into Android Studio and run the app. For instructions, see the Setup Guide for Android.

Key components

The following files contain the crucial code for this pose landmarking example application:

- PoseLandmarkerHelper.kt - Initializes the pose landmarker and handles the model and delegate selection.
- CameraFragment.kt - Handles the device camera and processes the image and video input data.
- GalleryFragment.kt - Interacts with `OverlayView` to display the output image or video.
- OverlayView.kt - Implements the display for the detected poses.

Setup

This section describes key steps for setting up your development environment and code projects specifically to use Pose Landmarker. For general information on setting up your development environment for using MediaPipe tasks, including platform version requirements, see the Setup guide for Android.

Attention: This MediaPipe Solutions Preview is an early release. Learn more.

Dependencies

The Pose Landmarker task uses the `com.google.mediapipe:tasks-vision` library. Add this dependency to the `build.gradle` file of your Android app:

```
dependencies {
    implementation 'com.google.mediapipe:tasks-vision:latest.release'
}
```

Model

The MediaPipe Pose Landmarker task requires a trained model bundle that is compatible with this task. For more information on available trained models for Pose Landmarker, see the task overview Models section.

Select and download the model, and store it within your project directory:

```
<dev-project-root>/src/main/assets
```

Specify the path of the model within the `ModelAssetPath` parameter. In the example code, the model is defined in the `PoseLandmarkerHelper.kt` file:

```
val modelName = "pose_landmarker_lite.task"
baseOptionsBuilder.setModelAssetPath(modelName)
```

Create the task

The MediaPipe Pose Landmarker task uses the `createFromOptions()` function to set up the task. The `createFromOptions()` function accepts values for the configuration options. For more information on configuration options, see Configuration options.

The Pose Landmarker supports the following input data types: still images, video files, and live video streams. You need to specify the running mode corresponding to your input data type when creating the task. Choose the tab for your input data type to see how to create the task.

Image Video Live stream

```
val baseOptionsBuilder = BaseOptions.builder().setModelAssetPath(modelName)
val baseOptions = baseOptionBuilder.build()

val optionsBuilder =
    poseLandmarker.poseLandmarkerOptions.builder()
        .setBaseOptions(baseOptionsBuilder.build())
        .setMinPoseDetectionConfidence(minPoseDetectionConfidence)
        .setMinTrackingConfidence(minPoseTrackingConfidence)
        .setMinPosePresenceConfidence(minposePresenceConfidence)
        .setNumPoses(maxNumPoses)
        .setRunningMode(RunningMode.IMAGE)

val options = optionsBuilder.build()
poseLandmarker = poseLandmarker.createFromOptions(context, options)
```

★ **Note:** If you use the live stream mode, you'll need to register a result listener when creating the task. The task calls the listener when it finishes processing a camera frame, with the detection result and the input image as parameters.

★ **Note:** If you use the video mode or live stream mode, Pose Landmarker uses tracking to avoid triggering palm detection model on every frame, which helps reduce latency.

The Pose Landmarker example code implementation allows the user to switch between processing modes. The approach makes the task creation code more complicated and may not be appropriate for your use case. You can see this code in the `setupPoseLandmarker()` function in the `PoseLandmarkerHelper.kt` file.

Configuration options

This task has the following configuration options for Android apps:

Option Name	Description	Value Range	Default Value
`runningMode`	Sets the running mode for the task. There are three modes: IMAGE: The mode for single image inputs. VIDEO: The mode for decoded frames of a video. LIVE_STREAM: The mode for a livestream of input data, such as from a camera. In this mode, resultListener must be called to set up a listener to receive results asynchronously.	`{IMAGE, VIDEO, LIVE_STREAM}`	`IMAGE`
`numposes`	The maximum number of poses that can be detected by the Pose Landmarker.	`Integer > 0`	`1`
`minPoseDetectionConfidence`	The minimum confidence score for the pose detection to be considered successful.	`Float [0.0,1.0]`	`0.5`
`minPosePresenceConfidence`	The minimum confidence score of pose presence score in the pose landmark detection.	`Float [0.0,1.0]`	`0.5`
`minTrackingConfidence`	The minimum confidence score for the pose tracking to be considered successful.	`Float [0.0,1.0]`	`0.5`
`outputSegmentationMasks`	Whether Pose Landmarker outputs a segmentation mask for the detected pose.	`Boolean`	`False`
`resultListener`	Sets the result listener to receive the landmarker results asynchronously when Pose Landmarker is in the live stream mode. Can only be used when running mode is set to `LIVE_STREAM`	`ResultListener`	`N/A`
`errorListener`	Sets an optional error listener.	`ErrorListener`	`N/A`

Prepare data

Pose Landmarker works with images, video files, and live video streams. The task handles the data input preprocessing, including resizing, rotation and value normalization.

The following code demonstrates how to hand off data for processing. These samples include details on how to handle data from images, video files, and live video streams.

Image Video Live stream

```
import com.google.mediapipe.framework.image.BitmapImageBuilder
import com.google.mediapipe.framework.image.MPImage

// Convert the input Bitmap object to an MPImage object to run inference
val mpImage = BitmapImageBuilder(image).build()
```

In the Pose Landmarker example code, the data preparation is handled in the `PoseLandmarkerHelper.kt` file.

Run the task

Depending on the type of data your are working with, use the `poseLandmarker.detect...()` method that is specific to that data type. Use `detect()` for individual images, `detectForVideo()` for frames in video files, and `detectAsync()` for video streams. When you are performing detections on a video stream, make sure you run the detections on a separate thread to avoid blocking the user interpose thread.

The following code samples show simple examples of how to run Pose Landmarker in these different data modes:

Image Video Live stream

```
val result = poseLandmarker.detect(mpImage)
```

Note the following:

- When running in the video mode or the live stream mode, you must provide the timestamp of the input frame to the Pose Landmarker task.
- When running in the image or the video mode, the Pose Landmarker task blocks the current thread until it finishes processing the input image or frame. To avoid blocking the user interpose, execute the processing in a background thread.
- When running in the live stream mode, the Pose Landmarker task returns immediately and doesn't block the current thread. It will invoke the result listener with the detection result every time it finishes processing an input frame.

In the Pose Landmarker example code, the `detect`, `detectForVideo`, and `detectAsync` functions are defined in the `PoseLandmarkerHelper.kt` file.

Handle and display results

The Pose Landmarker returns a `poseLandmarkerResult` object for each detection run. The result object contains coordinates for each pose landmark.

The following shows an example of the output data from this task:

```
PoseLandmarkerResult:
  Landmarks:
    Landmark #0:
      x            : 0.638852
      y            : 0.671197
      z            : 0.129959
      visibility   : 0.9999997615814209
      presence     : 0.9999984502792358
    Landmark #1:
      x            : 0.634599
      y            : 0.536441
      z            : -0.06984
      visibility   : 0.999909
      presence     : 0.999958
    ... (33 landmarks per pose)
  WorldLandmarks:
    Landmark #0:
      x            : 0.067485
      y            : 0.031084
      z            : 0.055223
      visibility   : 0.9999997615814209
      presence     : 0.9999984502792358
    Landmark #1:
      x            : 0.063209
      y            : -0.00382
      z            : 0.020920
      visibility   : 0.999976
      presence     : 0.999998
    ... (33 world landmarks per pose)
  SegmentationMasks:
    ... (pictured below)
```

The output contains both normalized coordinates (`Landmarks`) and world coordinates (`WorldLandmarks`) for each landmark.

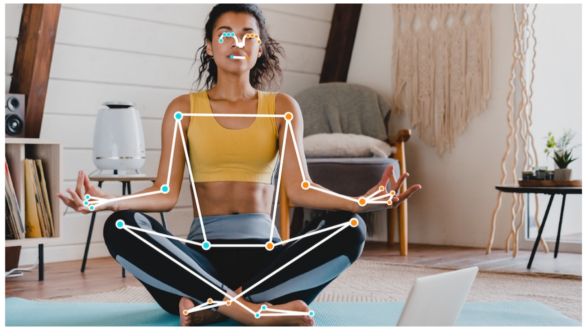
The output contains the following normalized coordinates (`Landmarks`):

- `x` and `y`: Landmark coordinates normalized between 0.0 and 1.0 by the image width (`x`) and height (`y`).
- `z`: The landmark depth, with the depth at the midpoint of the hips as the origin. The smaller the value, the closer the landmark is to the camera. The magnitude of z uses roughly the same scale as `x`.
- `visibility`: The likelihood of the landmark being visible within the image.

The output contains the following world coordinates (`WorldLandmarks`):

- `x`, `y`, and `z`: Real-world 3-dimensional coordinates in meters, with the midpoint of the hips as the origin.
- `visibility`: The likelihood of the landmark being visible within the image.

The following image shows a visualization of the task output:



The optional segmentation mask represents the likelihood of each pixel belonging to a detected person. The following image is a segmentation mask of the task output:

The Pose Landmarker example code demonstrates how to display the results returned from the task, see the `OverlayView` class for more details.

Was this helpful?

Send feedback

Last updated 2025-03-18 UTC.