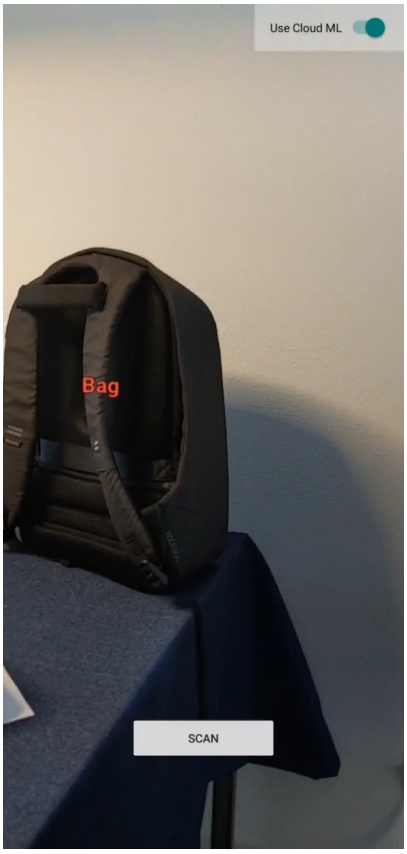
Use ARCore as input for Machine Learning models



You can use the camera feed that ARCore captures in a machine learning pipeline to create an intelligent augmented reality experience. The ARCore ML Kit sample (https://github.com/googlesamples/arcore-ml-sample) demonstrates how to use ML Kit (https://developers.google.cn/ml-kit) and the Google Cloud Vision API (https://cloud.google.com/vision) to identify real-world objects. The sample uses a machine learning model to classify objects in the camera's view and attaches a label to the object in the virtual scene.

The ARCore ML Kit sample (https://github.com/googlesamples/arcore-ml-sample) is written in Kotlin. It is also available as the **ml_kotlin** sample app in the ARCore SDK (https://github.com/google-ar/arcore-android-sdk/releases/) GitHub repository.

Us

AR

- A **CPU image stream** used for feature recognition and image processing. By default, the CPU image has a resolution of VGA (640x480). ARCore can be configured to use an additional higher resolution image stream, if required.
- A **GPU texture stream**, which contains a high-resolution texture, usually at a resolution of 1080p. This is typically used as a user-facing camera preview. This is stored in the OpenGL texture specified by `Session.setCameraTextureName()` (/ar/reference/java/com/google/ar/core/Session#setCameraTextureName-textureId).
- Any additional streams specified by `SharedCamera.setAppSurfaces()` (/ar/reference/java/com/google/ar/core/SharedCamera#setAppSurfaces-cameraId-surfaces).

CPU image size considerations

No additional cost is incurred if the default VGA-sized CPU stream is used because ARCore uses this stream for world comprehension. Requesting a stream with a different resolution may be expensive, as an additional stream will need to be captured. Keep in mind that a higher resolution may quickly become expensive for your model: doubling the width and height of the image quadruples the amount of pixels in the image.

It may be advantageous to downscale the image, if your model can still perform well on a lower resolution image.

Configure an additional high resolution CPU image stream

The performance of your ML model may depend on the resolution of the image used as input. The resolution of these streams can be adjusted by changing the current `CameraConfig` (/ar/reference/java/com/google/ar/core/CameraConfig) using `Session.setCameraConfig()` (/ar/reference/java/com/google/ar/core/Session#setCameraConfig(com.google.ar.core.CameraConfig)) , selecting a valid configuration from `Session.getSupportedCameraConfigs()` (/ar/reference/java/com/google/ar/core/Session#getSupportedCameraConfigs(com.google.ar.core.CameraConfigFilter)) .

Jav


```
        .setFacingDirection(CameraConfig.FacingDirection.BACK);
```

```
List<CameraConfig> supportedCameraConfigs =
    session.getSupportedCameraConfigs(cameraConfigFilter);

// Select an acceptable configuration from supportedCameraConfigs.
CameraConfig cameraConfig = selectCameraConfig(supportedCameraConfigs);
session.setCameraConfig(cameraConfig);
```

Retrieve the CPU image

Retrieve the CPU image using `Frame.acquireCameraImage()` (/ar/reference/java/com/google/ar/core/Frame#acquireCameraImage()). These images should be disposed of as soon as they're no longer needed.

Note: `Frame.acquireCameraImage()` (/ar/reference/java/com/google/ar/core/Frame#acquireCameraImage()) can throw `NotYetAvailableException` (/ar/reference/java/com/google/ar/core/exceptions/NotYetAvailableException) for several frames after session start, and for a few frames at a time while the session is running. Ensure that your application can handle this case.

JavaKotlin (#kotlin)
(#java)

```
Image cameraImage = null;
try {
  cameraImage = frame.acquireCameraImage();
  // Process `cameraImage` using your ML inference model.
} catch (NotYetAvailableException e) {
  // NotYetAvailableException is an exception that can be expected when
  // yet. The image may become available on a next frame.
} catch (RuntimeException e) {
  // A different exception occurred, e.g. DeadlineExceededException, Res
  // Handle this error appropriately.
  handleAcquireCameraImageFailure(e);
} finally {
  if (cameraImage != null) {
```

Process the CPU image

To process the CPU image, various machine learning libraries can be used.

- **ML Kit**: ML Kit provides an on-device Object Detection and Tracking (https://developers.google.cn/ml-kit/vision/object-detection) API. It comes with a coarse classifier built into the API, and can also use custom classification models (https://developers.google.cn/ml-kit/vision/object-detection/custom-models/android) to cover a narrower domain of objects. Use `InputImage.fromMediaImage` (https://developers.google.cn/android/reference/com/google/mlkit/vision/common/InputImage#fromMediaImage(android.media.Image,%20int)) to convert your CPU image into an `InputImage`.
- **Firebase Machine Learning**: Firebase provides Machine Learning APIs (https://firebase.google.cn/docs/ml) that work either in the cloud or on the device. See Firebase documentation on Label Images Securely with Cloud Vision using Firebase Auth and Functions on Android (https://firebase.google.cn/docs/ml/android/label-images).

Display results in your AR scene

Image recognition models often output detected objects by indicating a center point or a bounding polygon representing the detected object.

Using the center point or center of the bounding box that is output from the model, it's possible to attach an anchor to the detected object. Use `Frame.hitTest()` (/ar/reference/java/com/google/ar/core/Frame#hitTest(float,%20float)) to estimate the pose of an object in the virtual scene.

Important: `Frame.hitTest()` (/ar/reference/java/com/google/ar/core/Frame#hitTest(float,%20float)) expects coordinates in the **`VIEW`** (/ar/reference/java/com/google/ar/core/Coordinates2d#view) coordinate system. Since your model uses the CPU stream (#stream-types), your model will give results in the **`IMAGE_PIXELS`** (/ar/reference/java/com/google/ar/core/Coordinates2d#image_pixels) coordinate system. Use **`Frame.transformCoordinates2d()`** (/ar/reference/java/com/google/ar/core/Frame#transformCoordinates2d-inputCoordinates-inpu

to c

Cor

(#java)

```
// Suppose `mlResult` contains an (x, y) of a given point on the CPU ima
float[] cpuCoordinates = new float[] {mlResult.getX(), mlResult.getY()};
float[] viewCoordinates = new float[2];
frame.transformCoordinates2d(
    Coordinates2d.IMAGE_PIXELS, cpuCoordinates, Coordinates2d.VIEW, viev
// `viewCoordinates` now contains coordinates suitable for hit testing.
```

Use these **`VIEW`** coordinates to conduct a hit test and create an anchor from the result:

JavaKotlin (#kotlin)
(#java)

```
List<HitResult> hits = frame.hitTest(viewCoordinates[0], viewCoordinates
HitResult depthPointResult = null;
for (HitResult hit : hits) {
  if (hit.getTrackable() instanceof DepthPoint) {
    depthPointResult = hit;
    break;
  }
}
if (depthPointResult != null) {
  Anchor anchor = depthPointResult.getTrackable().createAnchor(depthPoir
  // This anchor will be attached to the scene with stable tracking.
  // It can be used as a position for a virtual object, with a rotation
  // estimated surface normal.
}
```

Performance considerations

Follow the following recommendations to save processing power and consume less energy:

- Do not run your ML model on every incoming frame. Consider running object

Next steps

- Learn about Best Practices for ML Engineering (https://developers.google.cn/machine-learning/guides/rules-of-ml).
- Learn about Responsible AI practices (https://ai.google/responsibilities/responsible-ai-practices/).
- Follow the basics of machine learning with TensorFlow course (https://tensorflow.google.cn/resources/learn-ml/basics-of-machine-learning).

Last updated 2022-02-01 UTC.