Recognizing Famous Places on Android

Seminar
Practical Applications of Multimedia Retrieval
Winter Semester 2016/2017

Tim Oesterreich, Romain Granger

Supervisors:
Haojin Yang, Christian Bartz
Prof. Dr. Christoph Meinel

April 2, 2017

Contents

1 Introduction

As deep learning is becoming an increasingly big influence in everyday applications, more and more focus is put into increasing its distribution to different platforms.
During the winter semester 2016/2017 a project seminar emerged, that laid focus on developing a mobile phone application for Google's Android operating system, that is capable of recognizing famous sights in big cities, like Berlin, from images taken with a smartphone camera.
Modern smartphones can, in some cases, outperform mid-range notebooks from a couple of years ago and, most interestingly, often have a dedicated GPU[1]. GPUs are usually used for deep learning because of their high parallelization capabilities.
Part of this seminar was the evaluation of a fairly recent deep learning framework called CNNDroid [1], which uses GPU acceleration for classification and promises substantial performance improvements compared to CPU classification.

[1] Graphics Processing Unit

2 CNNDroid

CNNDroid is an open source deep learning library for Android. It is able to execute convolutional neural networks, supporting most CNN layers used by existing desktop/server deep learning frameworks, namely Caffe, Theano and Torch. Supported layers, as of March 2017 are:

- Convolutional Layer
- Pooling Layer
- Local Response Normalization Layer
- Fully-Connected Layer
- Rectified Linear Unit Layer (ReLU)
- Softmax Layer
- Accuracy and Top-K Layer

Due to the library being open source, it is possible to add additional layers, such as batch normalization or sigmoid.
The library also supports a variety of customizations, like maximum memory usage, GPU or CPU acceleration and automatic performance tuning.

2.1 Setup and Integration into Android Project

CNNDroid is a source code library. That means that integration into an existing Android Project is fairly straight forward and doesn't require any third-party dependencies.
The only prerequisites are:

- A functional Android development environment (e.g. Android Studio[2])
- Android phone or emulator running at least Android SDK version 21.0 (Lollipop)

To integrate CNNDroid into the project, the repository has to be cloned from its GitHub[3] page. Inside the `CNNdroid Source Package`' folder are three folders: `java`, `rs` and `libs`.
The `java` and `rs` folders need to be copied into the projects `app/src/main/` directory, merging the `java` folders.
The `libs` folder has to be copied and merged into the `app/` directory.
For effective usage of CNNDroid, it needs read and write access to storage on the smartphone. Android policies require an application to request these permissions before the

[2]https://developer.android.com/studio/index.html
[3]https://github.com/ENCP/CNNdroid

app starts. These permission requests are specified inside the `AndroidManifest.xml` file. The following lines have to be added there before the `<application` section:

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

To cope with the high computational requirements, CNNDroid potentially needs a large amount of memory. Therefore, the heap has to be increased, which is done by adding

```
android::largeHeap="true"
```

inside the `<application` section.
Now, everything should be set up correctly and programming can commence.
After importing the package `network.CNNdroid`, the `CNNdroid` object is accessible, which can call the function `CNNDroid::classify`. This is the keyword to start execution of the trained model. The specification of the model is explained in the following section.

2.2 Structure of necessary CNNDroid Files

Of course, CNNDroid needs to know how to classify an input. For that it either uses converted binary layer files, created by a desktop deep learning framework, or internal conversion functions (e.g. for pooling or ReLU).
The layer BLObs[4] are generally used for more complicated layers, such as convolutional or fully connected layers, which have high dimensional variables (weight matrices). These files have to be converted into the MessagePack[5] format and put onto the smartphones storage.
The order in which the layers are executed, additional configurations and layers which do not need a BLOb file are defined inside the definition file. This file is used as a settings file for the `CNNdroid` classifier.
On creation of the `CNNdroid` object, the absolute path to this file has to be specified.
Configurations, other than the layer definitions are:

- the absolute path to the layer blob files
  (`root_directory: "/absolute/path/to/layer/files"`)
- the maximum amount of RAM the classifier should use on the smartphone
  (`allocated_ram: Amount_in_MB`)
- whether or not automatic performance tuning should be used
  (`auto_tuning: "on|off"`)
- whether or not classification should be GPU accelerated (if possible)
  (`execution_mode: "sequential|parallel"`)

[4] Binary Large Objects
[5] http://msgpack.org/

Layers are defined as shown in figure 1.

```
root_directory: "/sdcard/Download/berlin_sights_data/"
allocated_ram: 100
auto_tuning: "off"
execution_mode: "parallel"

layer {
  type: "Convolution"
  name: "conv1"
  parameters_file: "model_param_conv1.msg"
  pad: 2
  group: 1
  stride: 1
}
```

Figure 1: Excerpt of a CNNDroid definition file

Not essential for the execution of CNNDroid, but very helpful for processing the final results, is a labels file. This file should specify human-readable names for the extractable classes. The order inside this file should be mapped to the output order of the last layer (probably a fully-connected layer), i.e. the first value of the output array should correspond to the first class name inside the labels file. The class names should be newline-separated.

2.3 Training the model

We trained our deep learning model based on crawled images with the caffe framework. The training occurred on an HPI server (accessible under the IP 172.16.18.178 and username msws2016t1). Caffe is already installed there and a variety of crawled images, as well as a labels file with the paths to each image are available inside the `training_data` folder. Inside the same folder you can also find the `training_full.sh` script, which can be executed to start the training and create a model.
An example model can be found in the `training_data/berlin_sights_model` folder.
In order to test the model quickly, caffe provides an ipython notebook on their Github repository. This repository is checked out in the `caffe` folder. To start the notebook an ssh-tunnel has to be created to your local machine, e.g. using

```
ssh -N -f -L localhost:8888:localhost:8889 msws2016t1@172.16.18.178
```

Using a normal ssh connection to the server, the notebook can be executed with the command

```
ipython notebook -no-browser -port=8889
```

and viewed in a webbrowser on the local machine under the address `localhost:8888`.
The notebook itself provides possibilities to classify an image downloaded from a web address, view the output of specific layers and more.

2.4 Convert Trained Models into CNNDroid-compatible Format

As mentioned before, CNNDroid uses a format called MessagePack for the layer definitions. MessagePack is a binary serialization format for exchanging data, comparable to JSON[6], but other than JSON it is not human-readable. The omission of this constraint makes it possible to reduce size and increase parsing speed of the data stored inside. This is especially important due to the limited storage that most smartphones posses, especially compared to big servers, where deep learning is usually executed.
Figure 2 illustrates how MessagePack saves storage space by using types (JSON encodes everything as strings, so for example, 'true' uses four Bytes, while it only uses one Byte using MessagePack) and dropping control symbols, like the curly braces {}.

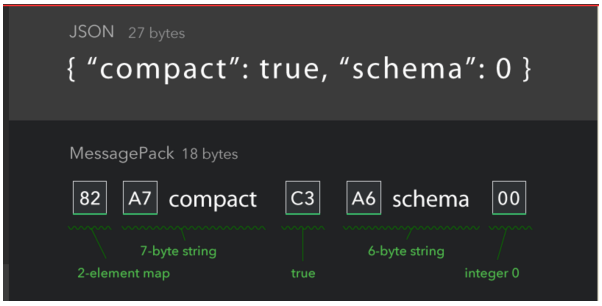


Figure 2: Comparison JSON and MsgPack

In order to convert models from the aforementioned desktop/server deep learning frameworks, CNNDroid provides conversion scripts, which can be found in the `CNNDroid/Parameter Generation Scripts/` folder. The script for Caffe is written in Python and requires the packages `numpy` and `msgpack` to be installed.
Three variables need to be defined.

- `MODEL_FILE` - The absolute path to the trained model
- `MODEL_NET` - The absolute path the deployment prototxt file, which specifies the parameters for the layers
- `SAVE_TO` - The saving path

The output are converted layer files in MessagePack format. The definition and label files have to be created manually. For the definition file, most of the content of the prototxt deployment file can be used. Only some parts of the syntax have to be adapted to a CNNDroid parsable format.

[6] JavaScript Object Notation

2.5 CPU vs GPU performance

For a comparison of the sequential and parallel mode of CNNDroid we used the Android emulator using 2 GB of RAM. We used the berlin sights classifier, provided in the berlin_sights_data folder.

layer	sequential time in ms	parallel time in ms
conv1	277	75
pool1	22	47
relu1	1	1
conv2	299	27
pool2	8	23
conv3	161	24
pool3	4	6
ip1	1	5
ip2	0	0
prob	0	0
sum	773	208

We can see that especially the more complicated convolutional layers profit from the additional speed of the GPU. Copying data from the RAM to GPU memory is usually very high, which is why the sequential mode can be faster for layers that do not do a lot of processing. In general, the parallel, GPU-accelerated processing speed is about three times faster than sequential, CPU-only processing.
The CNNDroid paper [1] states a 12-times speedup with the CIFAR10 dataset on a Samsung Galaxy Note 4.
Our test machine was a MacBook Pro with an Intel Iris Graphics 6100 GPU and a 2.9 GHz Intel Core i5 CPU using the Android Studio emulator.
Apparently, the emulator does not reach such a significant overall speedup as the one stated in the paper. The `conv2` layer reaches an 11-times speedup, which comes close, but in sum it only reaches a third of the expected performance.
This however can potentially have a variety of reasons. Our testing setup does not follow lab conditions. It is not possible to know how the Android Studio emulator uses the MacBooks resources. Additional overhead is to be expected when communicating with the hardware. Also the CPU in the MacBook is more powerful than the one inside the Samsung phone (using an NVidia Jetson-style dual CPU consisting of one Quad-core 1.9 GHz Cortex-A57 and one Quad-core 1.3 GHz Cortex-A53, usually only using the A57 for high-computation tasks), which would increase the speedup for the parallelized computation.
In theory the GPU of the MacBook should perform a lot better than the Samsung Galaxys GPU (which is capable of 48 parallel tasks, while the MacBooks GPU is capable of many hundred computations at the same time), it is however possible that the layer doesn't provide so many calculations, so that the full performance can not be used.

Overall, it is apparent, that the parallelization of the classification creates a clear speedup, which does however not reach that promised in the paper using our test setup.

3 Image Crawler

Classification of real-world images requires a very big dataset for the learning process beforehand. Google Streetview provides 360 degree images from many places in all German urban areas. Using sophisticated crawling techniques, a huge amount of image data can be extracted from these, so called, 'Photospheres'.
This chapter describes how we used the Google Streetview and other image APIs to crawl images from famous sights in Berlin.

3.1 Image Crawler

We provide a Google Streetview image crawling python script inside the `ImageCrawler/streetview_crawler` folder. This script takes a CSV[7] file as input, which specifies the parameters the crawler needs, such as position and viewing angle. Chapter 3.2 explains the contents of this file.
All requirements for the crawler are specified in the `requirements.txt` file and can be installed with the command `$ pip install -r requirements.txt`.
Given a location in longitude and latitude, the image crawler will automatically create jpeg-images from different viewing angles. These angles can either be specified or automatically generated by calculating the angle between two geo-coordinates. In the latter case, the script will generate images in five degree steps from 30 degrees to the left to 30 degrees to the right of the calculated viewing angle. If specified by hand, we use one degree steps from the starting to the end angle.
Because photospheres can change or get removed, the API uses the closest photosphere to the location provided. This might not be the location specified in the CSV file, which is why we also check the metainformation of each photosphere for the coordinates connected to the image and the status code. In case there is no image in a certain area, the status code tells us that and we don't need to go through all the requests.
To prevent Denial-of-Service attacks, the API uses a time- and request-based blocking system. That means, that an IP might be denied further downloads if it either makes request too fast or it reaches the maximum amount. To counter the first blockage, our script uses an exponential wait mechanism. If requests were made too quickly, it waits for a short while before retrying. If it still gets no result, the wait time is increased. This is repeated until the blockage is lifted.
For the second part, we provide the usage of a Google API key[8], which can be passed to the crawler script with the `--key {key}` command line switch. The API key enables the crawler to request up to 25'000 requests per day for free, with the option to increase the volume against money.

[7] Comma-Separated Values
[8] https://developers.google.com/maps/documentation/streetview/

3.2 Setup of viewing parameters

Inside the `places.csv` file, the crawler finds parameters for specified locations and their photospheres.
The necessary fields are:

- lat - the latitude of the photosphere in decimal format
- long - the longitude of the photosphere in decimal format
- fov - the field-of-view of the photosphere section in degrees
- pitch - the pitch of the camera in degrees (0 degrees equals parallel to the floor)
- starthead - the heading from where the camera sweep should start, or 0 if heading should be calculated automatically [optional, if buildingLat and buildingLong specified]
- endhead - the heading where the camera sweep should end, or 0 if heading should be calculated automatically [optional, if buildingLat and buildingLong specified]
- name - the name of object in the image; this name is being used as a file prefix for easier identification
- buildingLat - the latitude of the object in the image; used for automatic heading calculation [optional, if starthead and endhead specified]
- buildingLong - the longitude of the object in the image; used for automatic heading calculation [optional, if starthead and endhead specified]

The values are read line-by-line and have to be comma-separated.

3.3 State of Automation

The static image API we use only provides data about one specific photosphere. This enables us to get a fair amount of images from different viewing angles and different zoom levels for each position specified.
As mentioned before, we currently do a camera sweep from a start heading to an end heading, taking one image per degree. This creates on average 40 different viable images. Additionally, we scale each crawled image to 90% and 80% of its size, so we gain about 120 images for each line in the CSV file.
Unfortunately, this API doesn't provide the possibility to find out which further photospheres are nearby. For example, it is not possible to "drive down" a road by pressing the arrows, like on the Google Maps website.

As a future improvement, the JavaScript API could be used to find out which locations provide images and use this information to crawl more data with less manual work.
Also, to improve the quality of the images, it is possible to classify each image with our own sights classifier and find out if the object we search is actually in it and the image is not corrupted by other objects that we might also want to classify.

3.4 Current Limitations

Complete automation is not consistently possible at the moment. The static Image API we use for downloading the images does not work on the same database as the JavaScript API, that Google Maps uses. Therefore it is not easily possible to download many of the photospheres from the Streetview web service and results for the same geo-coordinates usually differ between the Image and the JavaScript API. There is currently a feature request open to resolve this issue[9].
Another limitation is the reported geo-coordinates in the metadata of the photospheres. These coordinates are usually provided by the camera that took them. GPS can be inaccurate up to 30-40 meters in bad cases. Especially if the image was taken very close to the object, this deviation can cause the automatic calculation of the heading to be wrong and not contain the object observed. This is why the CSV-file also provides the possibility to specify the starting and end heading.

3.5 Google Places API

For our project, we use several APIs from Google which help us to get additional information about places. Mainly, this is complementary information such as: Name and address of the place, phone number, website and rating from Google reviews.

3.5.1 Place Detection API

The place detection API allows us to discover places close to where the device is located. These are all places registered in Google including local businesses, points of interest, and geographic locations. This API will return a maximum of 10 probable places. Another very interesting use is that it can help us determinate if our prediction is relevant, by comparing our classification with the results from this function.

[9] https://code.google.com/p/gmaps-api-issues/issues/detail?id=10402&q=apitype%3AStreetView&sort=-stars&colspec=ID%20Type%20Status%20Summary%20Internal%20Stars

3.5.2 How to use it in Android

The first step is to get an API key from Google[10]. To get the API key, you need to register on the developer console from Google. This platform is used to access every Google API registered by your company.

After that, you need to declare your API key in the android manifest between the `<application` tag and the first `<Activity>` tag:

```
<meta-data
android:name="com.google.android.gms.version" android:value="@integer
    Google_play_services_version"/>
<meta-data
android:name="com.google.android.geo.APIKEY" android:value="API_KEY"/>
```

For the third step, you need to ask permissions in the android manifest for Internet access and GPS location.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

In your application, you need to declare a "GoogleApiClient" builder and add the APIs you want to use. In our case, and illustrated by the following snippet, we use the `PLACE_DETECTION_API` and `GEO_DATA_API`.

```
mGoogleApiClient = new GoogleApiClient.Builder(this)
                    .addApi(Places.PLACE_DETECTION_API)
                    .addApi(Places.GEO_DATA_API)
                    .enableAutoManage(this, GOOGLE_API_CLIENT_ID, this)
                    .build()
```

The final step is calling the API function. You need to use the `Places` package associated with the API you want to use. The Google API builder has to be passed to the function.

```
#For the PLACE DETECTION API
Places.PlaceDetectionApi.getCurrentPlace(mGoogleApiClient, null);

#For the GEO DATA API
Places.GeoDataApi.getPlaceById(mGoogleApiClient, placeId)
```

3.5.3 Limitations

Google provides a really useful and powerful service, but the requests are very limited. We can only have 1'000 free requests per 24 hours, which is probably not enough enough for a multi-user application. Therefore, Google provides a business API plan that can be adapted if needed. It gives access to 150'000 request per day and can be increased depending on the number of requests needed.

[10] https://developers.google.com/

3.6 Shutterstock, Flickr and Pinterest

Shutterstock[11], Flickr[12] and Pinterest[13] are public web application for hosting and sharing photos and images. For example, on Flickr nearly 7,000 photos are uploaded per minute. As these services are used by both professional and amateur photographers, we can find a lot of good quality images.

3.6.1 Example to get images URL

This is an example about how to get an image URL which can be downloaded, using the angularJS API package and store the resulting image in a mongoDB database.
First we create an API object:

```
var api = shutter.v2({
    clientId: 'Your public key',
    clientSecret: 'Your secret key',
});
```

Then we detail the search string we want to crawl. Here we choose the Brandenburg Gate.

```
var opts = {
    query: 'Brandenburg Gate',
    page: 1,
    per_page: 200,
    sort : 'popular'
};
```

Finally, we navigate through the JSON response to get the URL.

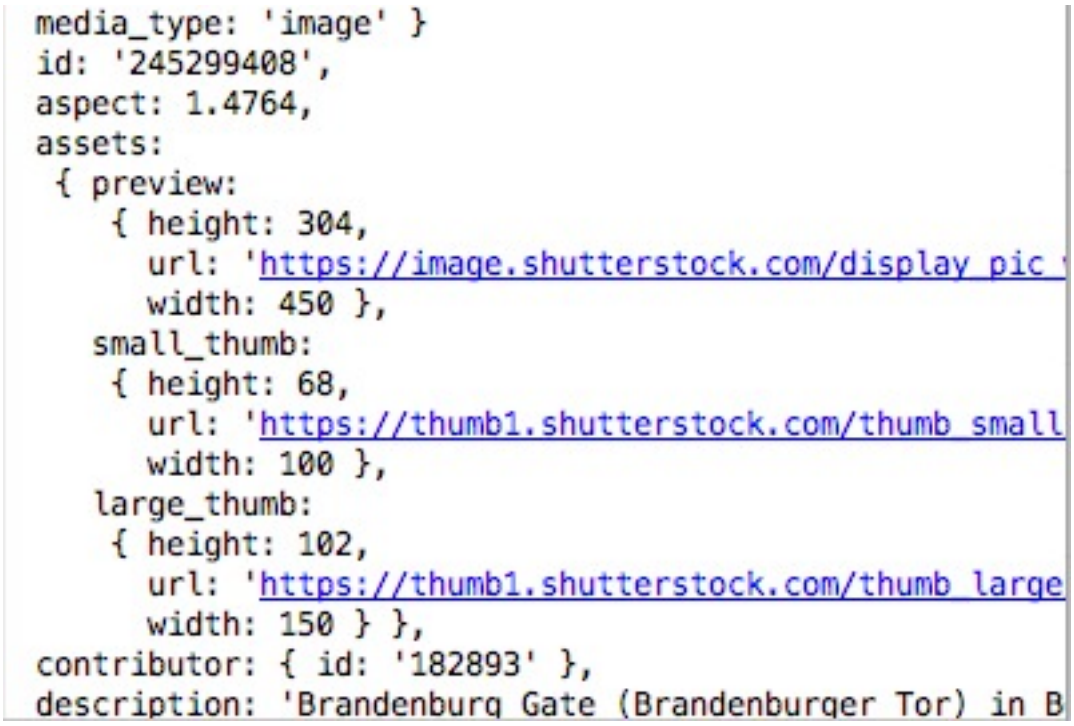
```
    api.image.search(opts, function(err, data) {
        if (err) throw err;

        console.log(data.data[0].id);
        var arrayLength = data.data.length;

        for (var i = 0; i < arrayLength; i++) {
            var obj = data.data[i];
            console.log(obj);
        }
```

[11] https://www.shutterstock.com/
[12] https://www.flickr.com/
[13] https://www.pinterest.com/

The response return will be as follows.



```
media_type: 'image' }
id: '245299408',
aspect: 1.4764,
assets:
 { preview:
    { height: 304,
      url: 'https://image.shutterstock.com/display_pic
      width: 450 },
   small_thumb:
    { height: 68,
      url: 'https://thumb1.shutterstock.com/thumb_small
      width: 100 },
   large_thumb:
    { height: 102,
      url: 'https://thumb1.shutterstock.com/thumb_large
      width: 150 } },
contributor: { id: '182893' },
description: 'Brandenburg Gate (Brandenburger Tor) in B
```

3.6.2 Limitations

These services are limited to 3600 call per hours. An API key can be blocked if the service detects an abusive use of the service.

4 PlaceRecognizer Application

After crawling the images and training a model, this section explains how we used these prerequisites to create an Android application that is capable of classifying images taken with the camera.

4.1 CNNDroid Integration/Image Classifier

The `ImageClassifier` class is the connection from CNNDroid to our application. It handles the creation of the CNNDroid classifier object as an asynchronous background task, the classification of a bitmap input and provides a function to output the accuracy of the result.
During creation of the CNNDroid object, the path to the model files (the layers in MessagePack format, the definition file and the labels file) has to be specified. The location of the files can be specified by setting the `rootpath` variable, which should point to the folder containing these files. Additionally, the name of the definition file has to be specified inside the background task.
The files have to be copied onto the Android device manually. If an emulator is used, the folder can simply dragged onto the emulated phone. Per default it is then saved at `/sdcard/Download/`. Old files are automatically overwritten.
If a physical phone is used, the files can either be copied using the operating systems file explorer (if supported) or using the command line tool `adb`[14], which comes with Android Studio. The command for copying from the host device to the android phone is

```
$ adb push '/local/path/' '/path/on/phone'
```

The `classifyImage` function takes a bitmap image as input. This image is converted into an array of shape `[n][numColorChannels][width][height]` by iterating over every pixel and splitting its RGB-value into its components. The dimensions of the array are defined as follows:

- n - Number of images to be classified. Because we currently only classify one image at a time, this is always 1.
- numColorChannels - If we classify on colored images, this is 3, with the first channel being the red, the second one the green and the last one the blue component. If the input image should be in grayscale there is only one channel and this part has to be adapted.
- width - width of the scaled input image
- height - height of the scaled input image

[14] Android Debug Bridge

The taken image is automatically scaled to fit the dimensions that the model expects. If the image dimensions of the trained model are changed, the variables `imageWidth` and `imageHeight` have to be changed as well.
Optionally, it is possible to use a mean file for improved classification results. The mean file has to be put into the folder specified by `rootpath` and the relevant code parts have to be uncommented. A python script that transforms a `.mean` file, for example generated by caffe, into MessagePack format is provided as well, called `binaryproto_to_msg.py` in our projects root directory.
The resulting input array is passed to the `CNNDroid::compute` function, which returns an array with shape `[n][numClasses]`. `input[0][0]` contains the predicted probability of the input image being the zeroth class.
The output array can now be passed to our `ImageClassifier::accuracy` function, which maps the accuracy to the corresponding label for the top k results. The returned value is a string showing this mapping, sorted from the most to least probable class.
Per default the `classify` function returns a `Classification` object, which contains the ID, probability and label of the best match found. The values can be extracted with `Classification::get_id()`, `Classification::get_label()` and `Classification::get_probability()`.

4.2 Real-Time Frame Capture

The `CameraFrameCapture` class is based on the Camera2 Android API. It is a specialized version of the Camera2Basic example[15].
The most important part is the preview of the camera image and the processing of each frame that comes out of it. For that, we create a CaptureRequest inside the `createCameraPreviewSession` function. This CaptureRequest needs to be rendered in order to access the image data. This is done by calling

```
mPreviewRequestBuilder.addTarget(surface);
mPreviewRequestBuilder.addTarget(mImageReader.getSurface());
```

The first part makes the image visible inside the application. The second part passes the image to our imageReader.
The imageReader has an `OnImageAvailableListener` callback, which is executed every time the imageReader receives a new image (i.e. every frame). The image provided by the API is in YUV-format[16]. To process the pixels, we need to convert them into RGB. This is done by saving the byte array into a YuvImage object, which has the ability to convert itself into jpeg, which contains RGB-values and can be decoded into a bitmap object. This bitmap object can now be passed to our classifier.
Right now, we only classify the image and use the debug information to evaluate the result. A function `getCurrentFrame` is also provided, which is a getter function for the

[15] https://github.com/googlesamples/android-Camera2Basic
[16] https://en.wikipedia.org/wiki/YUV

bitmap image, so that other classes, like the `MainActivity` class or the `ImageClassifier` can use the image.

4.3 GPS Logger

The `GPSTracker` class provides the possibility to use the smart phones GPS receiver to get the current location. The `GPSTracker::getLocation()` function handles permission requests to the user, who has to allow our application to use location services and uses Androids `LocationManager` to receive the current location.
The return value is a `Location` object, which holds a lot of information about the GPS result, like latitude and longitude, heading, altitude, accuracy and speed. Because we only care about the position for now, we only provided a `GPSTracker::getLatitude()` and `GPSTracker::getLongitude()` function, which return their respective value as a decimal value.

4.4 Wikipedia Parser

When the CNNDroid classifier calculated an image class, we now want to get information about the place. Wikipedia is one of the largest source of information and has a very big and active community, which regularly updates the articles. Also, the list of features we can grab from Wikipedia is long: Descriptions, images, literature, etc.. Additionally, compared to other APIs, the Wikipedia API presents three main advantages:

- The information is provided by making a simple URL request
- There use no usage limitations
- It provides an easily parsable JSON response

The Wikipedia API is composed of a principal URL which is `https://en.wikipedia.org/w/api.php`. This lets your change the language of the response easily by changing the subdomain of the URL. For example, to get a French JSON content, you can call `https://fr.wikipedia.org/w/api.`. This principal URL will be enriched with parameters, like the `format` (xml, json, html) or the `titles` to get information about a specific page. The second part of this section will clarify which parameters we use to call the Wikipedia API.

Our application uses two classes:

4.4.1 HttpHandler.java

The first class, "HttpHandler.java" creates a temporary byte array from the URL and also catches errors to check if the request is correct and returned a value. At first we use the http GET method to open a connection.

```
try {
    URL url = new URL(reqUrl);
    HttpURLConnection conn = (HttpURLConnection)
    url.openConnection();
    conn.setRequestMethod("GET");
```

Then we will catch errors, for the following cases:Incorrect input URL, Protocol exception, Input/Output error during the use of the InputStream() function, or if the response is an empty array of bytes.

```
} catch (MalformedURLException e) {
        Log.e(TAG, "MalformedURLException: " + e.getMessage());
    } catch (ProtocolException e) {
        Log.e(TAG, "ProtocolException: " + e.getMessage());
    } catch (IOException e) {
        Log.e(TAG, "IOException: " + e.getMessage());
    } catch (Exception e) {
        Log.e(TAG, "Exception: " + e.getMessage());
```

4.4.2 GetWiki.java

This is a public class using an asynchronous task. Our classifier will map a name to the image taken by the user, for example "Brandenburg Gate" or "Fernsehturm Berlin". This label will be used as the input string variable for our GetWiki class.
First, the class is called from the mainActivity class, with the following statement:

```
wikipediaInfos = new GetWiki().execute("classLabel")
```

"classLabel" is a string variable, which is dynamically update with the class given by the classifier. So it is also important to give a reliable class label regarding Wikipedia when the model is trained, because this label will be used as a parameter in the URL which is called:

```
String urlTitle = strings[0];
String url = "https://en.wikipedia.org/w/api.php?" +
             "format=json&action=query&prop=extracts&exintro=&explaintext=&titles="
                 +
             urlTitle;
```

The response will be a JSON array that we can parse to extract the information we need. An example of a response looks like the following:

```
{"batchcomplete":"","query":{"pages":{"156604":{"pageid":156604,"ns":0,"title":
    "Brandenburg Gate","extract":"The Brandenburg Gate (German: Brandenburger Tor)
    is an 18th-century neoclassical monument in Berlin, and one of the best-known
    landmarks of Germany. It is built on the site of a former city gate that
    marked the start of the road from Berlin to the town of Brandenburg an der
    Havel....\n"}}}}
```

We truncated the response to keep only the parts we actually use, like the title and the description. The response contains of several JSON objects. We use the JsonObject package provided in the Android API 25, to parse the JSON response. In our example, we would like to get all informations contained in the `156606:{` object:

```
JSONObject jsonObj = new JSONObject(jsonStr);
JSONObject query = jsonObj.getJSONObject("query");
```

```
JSONObject pages = query.getJSONObject("pages");
Iterator<String> keys = pages.keys();
String pageId= keys.next();
JSONObject page = pages.getJSONObject(pageId);
String title = page.getString("title");
String extract = page.getString("extract");
```

First, we put the response into a JsonObject variable. Then we can use the string parameters to navigate through the JSON file. If we have an object that is not static, like the ID of the page, we use an Iterator `keys` which will return the value for this JSON Object. In the example, it will give us the pageId (156606) as a string. Then we just have to say that we want to go to the next value of the JSON array using keys.next(). Finally, we get the string value of the response and store them into variables.

4.5 Text to speech

To enhance the user experience of our app, as it is used to recognize famous places, we had a look into how to allow the user to play an audio description of the places description while they are watching it. The Android SDK provides a useful package for this:

```
import android.speech.tts.TextToSpeech
```

which has built in methods to parse a string value and read it as an audio track. It can easily be implemented with few lines of code, following these steps:

First, declare a TTS (text to speech) object, and set the attributes. Many attributes can be changed, like the language, the speed of the audio or the voice type. The following snippet shows how to create a textToSpeech object talking UK english.

```
t1=new TextToSpeech(getApplicationContext(), new TextToSpeech.OnInitListener() {
            @Override
            public void onInit(int status) {
                if(status != TextToSpeech.ERROR) {
                    t1.setLanguage(Locale.UK);
                }
            }
        });
```

We can now pass a string, which should be converted into audio, to this object. In the following snippet we get the Wikipedia description of Brandenburg Gate from our Wikipedia API parser (see Section 4.4).

```
String toSpeak = null;
try {
toSpeak = new GetWiki().execute("Brandenburg_Gate").get().description;
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    }
```

Finally, you can use the speak() method from your object, with the queue mode (`QUEUE FLUSH`), which means that media to be played is dropped and replaced by the new entry each time you call this method.

```
t1.speak(toSpeak, TextToSpeech.QUEUE_FLUSH, null, "");
```

4.6 Firebase API

Firebase is a Google service which provides a realtime database and backend as a service. The service is very well integrated into Android. The data is stored on Firebases cloud and the Firebase SDK for android comes up with several method to store and synchronize data in real time.

4.6.1 How we use it

First, we declare the Firebase service in the `build.gradle` file.

```
compile 'com.google.firebase:firebase-core:10.0.1'
compile 'com.google.firebase:firebase-database:10.0.1'
```

Then, like the GooglePlaceAPI, we need to setup an object of type database to start the connection with our database instance.

```
private FirebaseAuth mAuth;
private FirebaseAuth.AuthStateListener mAuthListener;
DatabaseReference database = FirebaseDatabase.getInstance().getReference();
```

The Firebase structure uses paths to store the data. For example, you will have the top level path "users", which contains folders for each user of the app. Inside these directories the features are stored. To store or read data, the method will look like this:

```
database.child("users").child(userId).child("places").push("location");
```

You need to define the levels you want to have access to. Here we first access the "users" node, then getting the current user ID, and we upload the locations details of his visit into the "places" node.

4.7 How to setup project in Android Studio

The project should work without any additional work in Android Studio. Simply import it into the IDE, setup an Android emulator or plug in an android phone running at least Android SDK 24.0 and press the "Run" button.

5 Outlook

We have several ideas to improve our application.

Right now the application is very basic. It only contains two buttons which call the most essential functions and show the most essential information. Improvements of the visualization, like showing the camera image in full screen and switching to the information pane on getting a result, providing an option page, for example for switching language or user or possibilities to receive further information about a recognized place by showing links to Wikipedia or Google could add a lot of user experience to the application.
Because our application is aimed at international tourists, it is a good idea to localize it. This can be achieved by providing translation resource files, which cold look like this:

```
#/values/strings.xml:
<resources>
    <string name="hello_world">Hello World!</string>
</resources>

#/values-es/strings.xml
<resources>
    <string name="hello_world">¡Hola Mundo!</string>
</resources>
```

Also for the Wikipedia parser, we can get the phone language with a simple line of code

```
Locale.getDefault().getDisplayLanguage();
```

Lastly, a naive solution for improving the recognition results, especially if many objects are nearby is pre-filtering the possible results using the current position and only allowing results that are within a certain radius. A more complex approach would be to include the coordinates into the training itself. That means that a result is not only based on computing pixel values, but also considers the current position. This would provide an additional degree of confidence into the calculated result.

References

[1] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In *Proceedings of the 2016 ACM on Multimedia Conference*, MM '16, pages 1201–1205, 2016.