Capstone Project

Machine Learning Engineer Nanodegree

Martin Bede

April 6th, 2016

Definition

Project Overview

Vision loss is a problem that affects people all around the world. As computers are getting better at understanding images due to advances in computer vision, the concept of a virtual assistant for the blind that could read text[1], identify/spot objects[2], or even describe a whole scene in natural language[3] is becoming increasingly realistic.
In this project, I created an Android application capable of reading aloud everyday text (e.g. product labels, price tags, text on clothing). The application uses a classifier trained using the MS-COCO[4] and COCO-Text[5] datasets to look for text and uses Google Cloud Vision to extract the text.
The project was inspired by this Reddit thread.

Problem Statement

The goal is to create a general text-reader running on Android smartphones; the tasks involved are the following:

1. Download and preprocess the MS-COCO and COCO-Text data
2. Train a classifier that can determine if an image contains text
3. Make the classifier run on Android
4. Make the app extract the text (if any) using Google Cloud Vision
5. Make the app speak the extracted text aloud

The final application is expected to be useful for reading product labels, price tags, and other kinds of short, printed text.

Metrics

Accuracy is a common metric for binary classifiers; it takes into account both true positives and true negatives with equal weight.

$$accuracy = \frac{true\ positives + true\ negatives}{dataset\ size}$$

[1] Jaderberg, Max et al. "Reading text in the wild with convolutional neural networks." *International Journal of Computer Vision* 116.1 (2016): 1-20.
[2] "BlindTool" <https://play.google.com/store/apps/details?id=the.blindtool&hl=en> (2016).
[3] Devlin, Jacob et al. "Language models for image captioning: The quirks and what works." *arXiv preprint arXiv:1505.01809* (2015).
[4] Lin, Tsung-Yi et al. "Microsoft coco: Common objects in context." *Computer Vision–ECCV 2014* (2014): 740-755.
[5] Veit, Andreas et al. "COCO-Text: Dataset and Benchmark for Text Detection and Recognition in Natural Images." *arXiv preprint arXiv:1601.07140* (2016).

This metric was used when evaluating the classifier because false negatives and false positives both erode the user experience:

- ❖ False negatives result in either a longer delay between the user pointing the camera text and device speaking the text (“**processing delay**”) or in the worst case, completely prevent the application from reading said text
- ❖ On the other hand, false positives make the application try to extract text from images that don’t contain any. This results in unnecessary computations on the remote server, which can be both costly and slow the application down. In the worst case, this might also result in the application reading gibberish.

Processing delay (defined above) is also a metric that has a big effect on the user experience. It can be broken into two components, as the image processing is done in two steps:

$$processing\ delay \approx classification\ delay + extraction\ delay$$

- ❖ The classification delay is the time it takes for the classifier to detect text; it is important by itself, as the application provides feedback to the user immediately after it detects text.
- ❖ The extraction delay is the time it takes for the application to start speaking after text was detected by the classifier.

Analysis

Data Exploration

The COCO (Common Objects in Context) dataset has hundreds of thousands of richly annotated images; the annotations are not described here because only the images are used. More accurately, only a subset of the images are used, the 2014 training images, as the COCO-Text 1.0 dataset has annotations only for that particular subset. Of the 82,783 images, 63,686 contain text; altogether there are 173,589 text instances, which is more than enough to train the classifier. The images are colored and have around 600 * 400 pixels each.

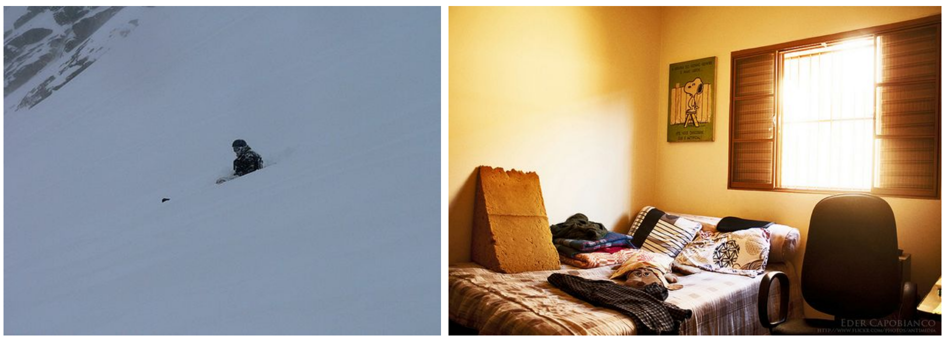


Fig. 1 Images from the MS-COCO dataset

The text annotations of the COCO-Text dataset have the following fields:

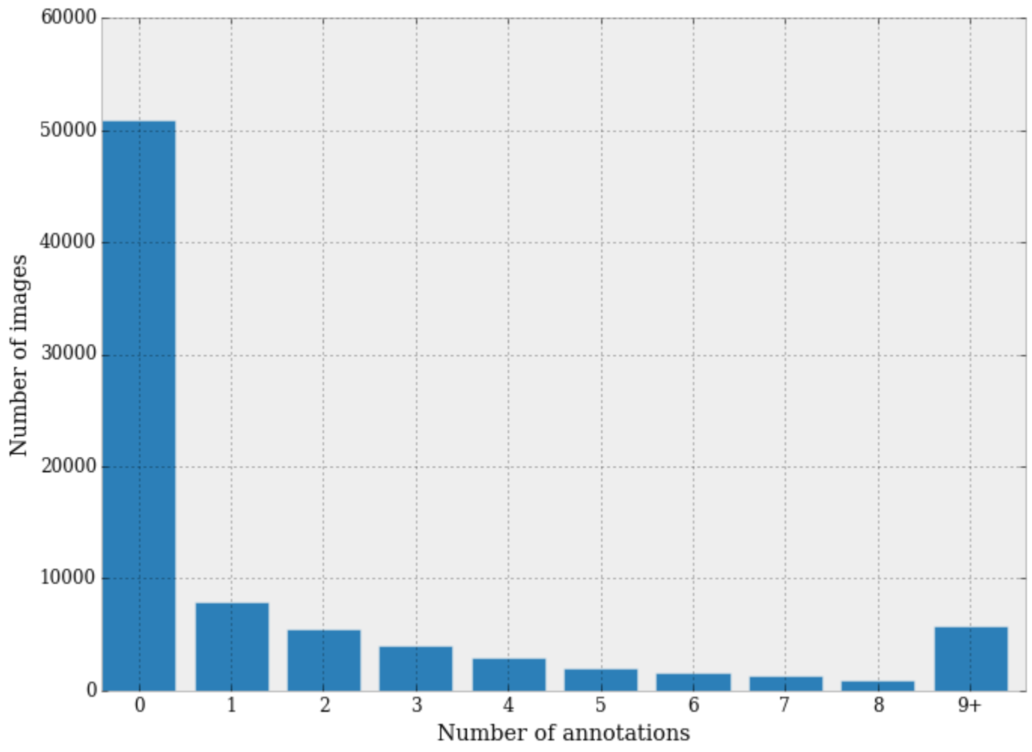
- “**utf8_string**”: the text itself (string)
- “**bbox**”: a bounding box around the text, in the form of [x, y, width, height] (integers)
- “**language**”: the language of the text; either “english”, “not english”, or “na” (string)
- “**legibility**”: the readability of the text; either “legible” or “illegible” (string)
- “area”: the area of the bounding box (float)
- “class”: the type of the text; either “machine printed, “handwritten”, or “others” (string)
- “image_id” (integer)
- “id” (integer)

As it can be suspected based on the above fields, some of the annotations are either illegible or not in English (or both), which means they must be discarded during preprocessing.

Exploratory Visualization

The plot below shows how the legible, English text annotations are distributed among the images. This is helpful for predicting how balanced the classes will be after the images are segmented (see the Data Preprocessing section).

Fig. 2 A plot showing how the legible, English text annotations are distributed among the images. Note that the distribution is right-tailed. (The 9+ column is large only because it contains the sum of all of the other columns which were cut off from the plot.) If the distribution was left-tailed, that would suggest that the annotations are mostly assigned to a few of the images, which could result in overfitting.

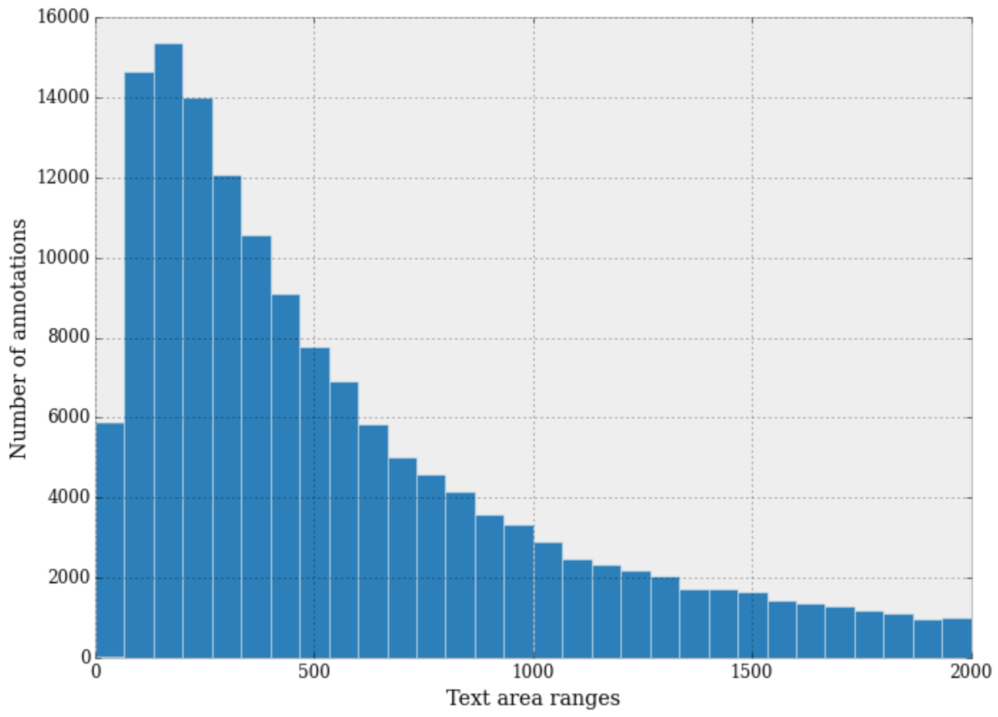


It can be seen that:

- ❖ The majority (50,000) of the images don't have text annotations.
- ❖ About two-thirds of the images that do have annotations have less than 5.

Fig. 3 The following plot shows how the areas of the bounding boxes of the same annotations are distributed.

This information can be helpful when trying to decide how big the image segments should be; if the area of an annotation is larger than the area of a segment, then the annotation will be surely split apart, which should be avoided for most of the annotations.



When interpreting the areas of the bounding boxes, it's helpful to take the square root of the values and keep in mind that most of the images have a resolution around 600 * 400.

It can be seen that most of the bounding boxes are quite small; this indicates that the annotations are usually just a few words long and that usually the text is in the background.

Algorithms and Techniques

The classifier is a Convolutional Neural Network, which is the state-of-the-art algorithm for most image processing tasks, including classification. It needs a large amount of training data compared to other approaches; fortunately, the COCO and COCO-Text datasets are big enough. The algorithm outputs an assigned probability for each class; this can be used to reduce the number of false positives using a **threshold**. (The tradeoff is that this increases the number of false negatives.)

The following parameters can be tuned to optimize the classifier:

- ❖ Classification **threshold** (see above)
- ❖ Training parameters
    - ➢ Training length (number of epochs)

    - Batch size (how many images to look at once during a single training step)
    - Solver type (what algorithm to use for learning)
    - Learning rate (how fast to learn; this can be dynamic)
    - Weight decay (prevents the model being dominated by a few “neurons”)
    - Momentum (takes the previous learning step into account when calculating the next one)
- Neural network architecture
    - Number of layers
    - Layer types (convolutional, fully-connected, or pooling)
    - Layer parameters (see links above)
- Preprocessing parameters (see the Data Preprocessing section)

During training, both the training and the validation sets are loaded into the RAM. After that, random batches are selected to be loaded into the GPU memory for processing. The training is done using the Mini-batch gradient descent algorithm (with momentum).
In contrast to this, inference (that is, inference in the Android app) is done using only the CPU, because TensorFlow doesn’t support smartphone GPUs.

Benchmark

To create an initial benchmark for the classifier, I used DIGITS (a web interface to the Caffe deep learning library, to try multiple architectures. The “standard” LeNet architecture (**Fig. 5**) achieved the best accuracy, around 0.8.
I couldn’t find a similar project that didn’t require special hardware, so the processing delay and classification delay benchmarks had to be created without actual data:

- For the classification delay, my goal was going below 3 seconds, optimally below 200 ms.
- For the overall processing delay, my goal was reaching a delay below 6 seconds, optimally below 500 ms.

The above two values were determined by asking a few people, rounding the responses, and taking the most frequent values.

Methodology

Data Preprocessing

The preprocessing done in the “Prepare data” notebook consists of the following steps:

1. The list of images is randomized
2. The images are divided into a training set and a validation set
3. The images are split into square shaped segments; random noise is used for padding
4. Each of the segments gets a label, which is “text” if the overlap between the segment and one of the annotations[6] is greater than a **threshold**, and “no-text” otherwise

There are also some preprocessing steps which are done as the images get loaded into memory before training:

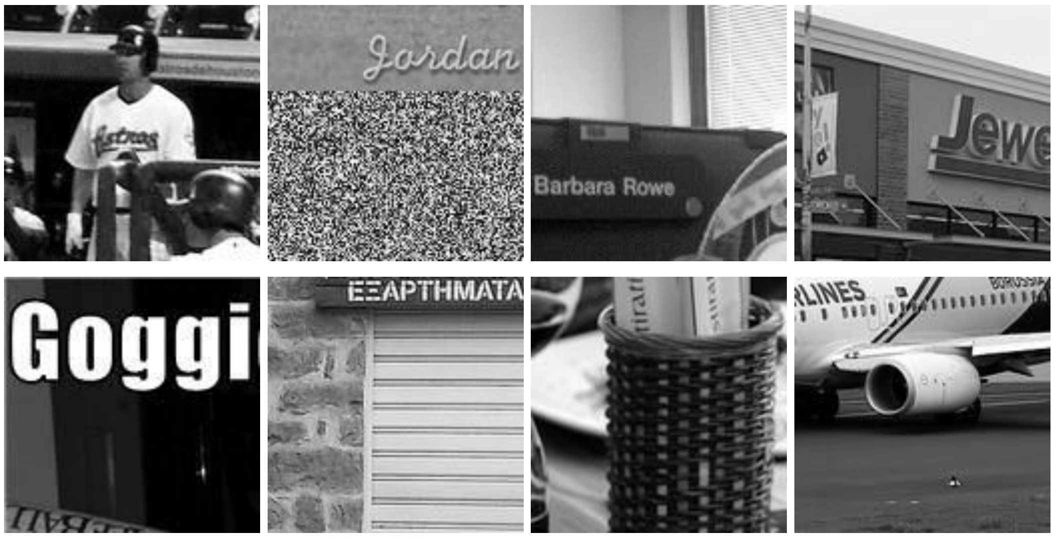
1. The images are converted to grayscale

[6] Precisely, one of the annotations that is both legible and in English

2. The pixel values get transformed to 16-bit floats
3. The mean pixel value is subtracted
4. The pixel values get divided by the standard deviation of the pixel values

Note that both the mean pixel value and the standard deviation are constants, and were determined earlier by sampling the training data.

Fig. 4 Segments from the "text" class produced with the final parameter settings. The segment size is 128 px, the overlap threshold is 500 px^2.



During inference (on Android), the latter steps are the same. However, the images are scaled before segmentation using a different method; instead of padding with random values, the original image is stretched. This does not affect the resulting segments significantly because the size difference between the original and the stretched image is small.
The preprocessing process has the following adjustable parameters:

- ❖ Segment size
- ❖ Overlap **threshold**
- ❖ Upper limit on the number of training images
- ❖ Upper limit on the number of validation images

Implementation

The implementation process can be split into two main stages:

1. The classifier training stage
2. The application development stage

During the first stage, the classifier was trained on the preprocessed training data. This was done in a Jupyter notebook (titled "Create and freeze graph"), and can be further divided into the following steps:

1. Load both the training and validation images into memory, preprocessing them as described in the previous section
2. Implement helper functions:
   a. get_batch(...): Draws a random sample from the training/validation data

b. fill_feed_dict(...): Creates a feed_dict, which is a Python dictionary that contains all of the data required for a single training step (a batch of images, their labels, and the learning rate)

3. Define the network architecture and training parameters
4. Define the loss function, accuracy
5. Train the network, logging the validation/training loss and the validation accuracy
6. Plot the logged values
7. If the accuracy is not high enough, return to step 3
8. Save and freeze the trained network

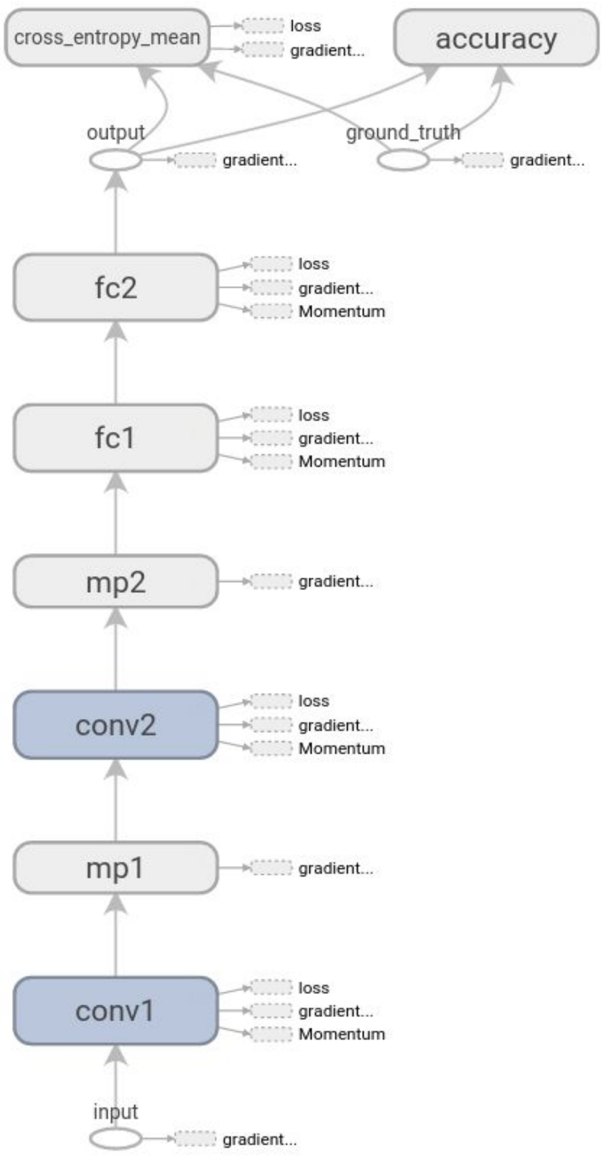


Fig. 5 The illustration on the right shows the computational graph, which includes the network architecture and also the variables that are only present during training.
The acronyms can be read as follows:

- **FC**: Fully connected layer
- **Conv**: Convolutional layer
- **MP**: Max pooling layer

The following can be seen by looking at the graph:

- The loss function is mainly composed of the mean cross entropy error.
- Large weights and biases of all four layers are penalized by using weight decay. (The weights are added to the loss function after they are multiplied by their specific weight decay rates.)

The application development stage can be split into the following steps:

1. Copy and compile the TensorFlow Android demo
2. Modify the original application to run the network trained in the previous stage (the most heavily modified files are listed in the readme)
3. Implement the call to the Cloud Vision API
4. Add TTS function

Refinement

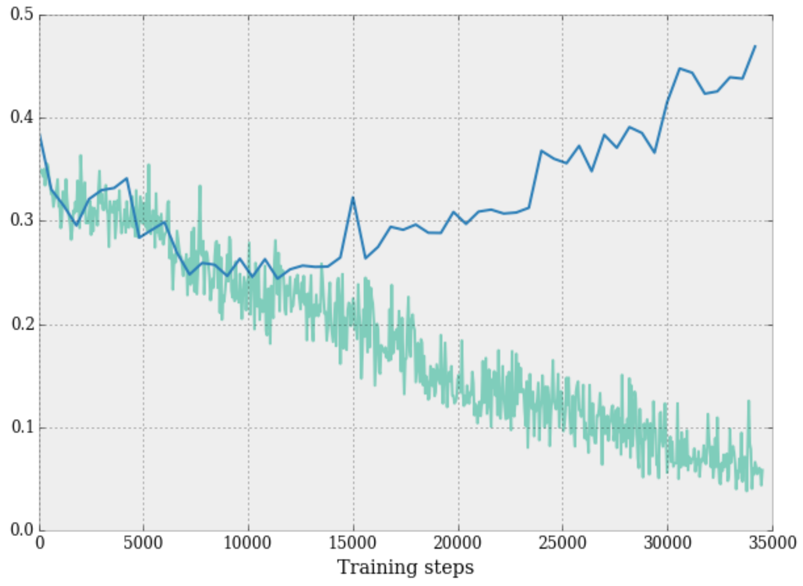
As mentioned in the Benchmark section, the LeNet architecture trained with Caffe achieved a good accuracy, around 80%. (This is considered good because around 20% of the positive training examples contain only a single character, or just some portion of a character, and as such are essentially false positives.)

To get the initial result, this architecture was ported to TensorFlow; the result was an accuracy around 70%. This was improved upon by using the following techniques:

- Dynamic learning rate: whenever the loss function stopped decreasing, a learning rate drop was added
- Weight decay: when overfitting was detected (the training and validation losses diverged too much), the weight decay rate was increased
- Adding dropout to a layer: dropout randomly drops weights in the layer it's applied to during training and scales the weights so that the network keeps working during inference. This was later undone because the Android build of TensorFlow doesn't support it.

The final TensorFlow model was derived by training in an iterative fashion, adjusting the parameters (e.g. learning rate, weight decay ratios) based on plots like the one below. The final model has an accuracy of 77%.

Fig. 6 A plot of the training/validation losses. Divergence indicates overfitting, which can be addressed by intensifying the weight decay, adding dropout, or reducing the model complexity (e.g. reducing the number of layers), among other techniques.



Results

Model Evaluation and Validation

During development, a validation set was used to evaluate the model.

The final architecture and hyperparameters were chosen because they performed the best among the tried combinations.
For a complete description of the final model and the training process, refer to **Figure 5** along with the following list:

- ❖ The shape of the filters of the convolutional layers is 5*5.
- ❖ The first convolutional layer learns 32 filters, the second learns 64 filters.
- ❖ The convolutional layers have a stride of 2, so the resolution of the output matrices is half the resolution of the input matrices.
- ❖ Like the convolutional layers, the pooling layers halve the resolution too.
- ❖ The weights of the convolutional layers are initialized by sampling a normal distribution with a standard deviation of 10^{-4}.
- ❖ The weights of the first and second fully connected layer are initialized by sample a normal distribution with a standard deviation of 0.04 and 0.1, respectively.
- ❖ The first fully connected layer has 512 outputs, the second 2. (The outputs of the latter correspond to the two classes, “text” and “no-text”.)
- ❖ The training runs for 36,000 iterations.
- ❖ The learning rate is multiplied by 0.003 when the number of iterations reaches 18,000.

To verify the robustness of the final model, a test was conducted using everyday household objects (**Fig. 7**). The following observations are based on the results of the test:

- ❖ The classifier can reliably detect high contrast text
- ❖ Blurry text is ignored, the camera must be focused for successful detection
- ❖ False positives are rare but present
- ❖ The classifier can detect text that is too far away for the text extractor

Justification

Using a Galaxy Note 4 on a 4G network/Wi-Fi, I got the following results:

- ❖ The classification delay is about 3 seconds, which is about the same as that of the benchmark
- ❖ The processing delay is around 5 seconds, which is better than that of the benchmark
- ❖ The per-image classification accuracy[7] is higher than 90%[8]

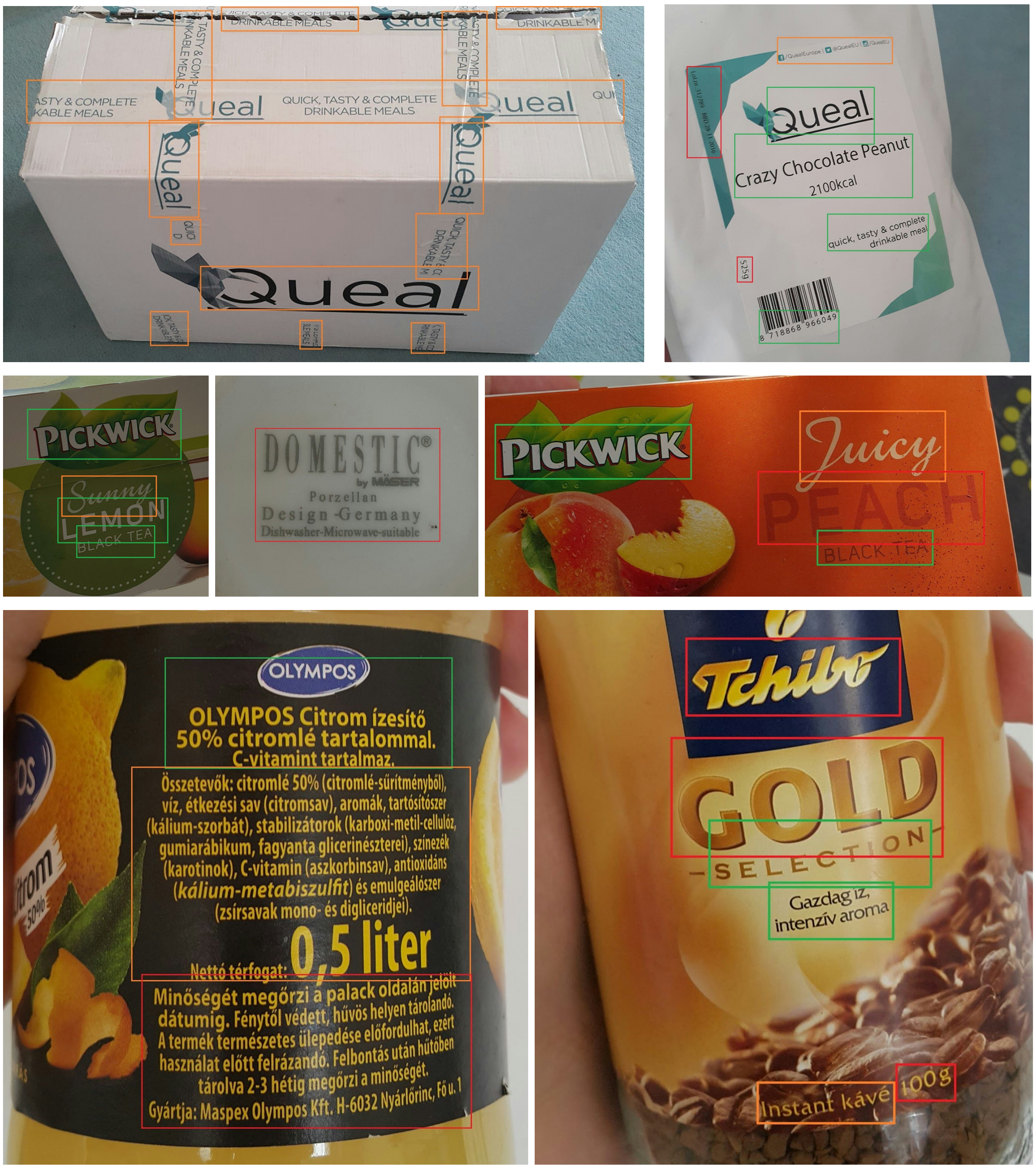
To understand how successful the final application is, it’s also important to know the overall text extraction performance, which is illustrated by **Figure 7**.
It can be seen that the application is useful for reading labels, but also that it can get confused by hard to read text (such as that on the cardboard box), and it can miss text that is too low-contrast compared to its background.
In summary, the application is useful in a limited domain, but to solve the bigger problem (giving visually impaired people access to written information), different hardware will have to be used (see the Improvement section)

[7] The per-image accuracy is used here because it affects the user experience more directly than the per-segment accuracy.
[8] The reason that value isn’t more precise is that I only had a few household items with text on them to test with

Conclusion

Free-Form Visualization

Fig. 7 Examples of text detected/ignored by the application. Each of the objects was tested with 3 times; the green boxes mark text that was read successfully at least 2 times. Orange means at least some words were successfully read at least one time. Red means the text was consistently ignored.



QUICK, TASTY & COMPLETE DRINKABLE MEALS
Queal
Crazy Chocolate Peanut
2100kcal
quick, tasty & complete drinkable meal
8 718868 966049
PICKWICK
Sunny LEMON
BLACK TEA
DOMESTIC® by MÄSER
Porzellan
Design Germany
Dishwasher-Microwave-suitable
PICKWICK
Juicy PEACH
BLACK TEA
OLYMPOS
OLYMPOS Citrom ízesítő 50% citromlé tartalommal. C-vitamint tartalmaz.
Összetevők: citromlé 50% (citromlé-sűrítményből), víz, étkezési sav (citromsav), aromák, tartósítószer (kálium-szorbát), stabilizátorok (karboxi-metil-cellulóz, gumiarábikum, fagyanta glicerinészterei), színezék (karotinok), C-vitamin (aszkorbinsav), antioxidáns (kálium-metabiszulfit) és emulgeálószer (zsírsavak mono- és digliceridjei).
Nettó térfogat: 0,5 liter
Minőségét megőrzi a palack oldalán jelölt dátumig. Fénytől védett, hűvös helyen tárolandó. A termék természetes ülepedése előfordulhat, ezért használat előtt felrázandó. Felbontás után hűtőben tárolva 2-3 hétig megőrzi a minőségét.
Gyártja: Maspex Olympos Kft. H-6032 Nyárlőrinc, Fő u. 1
Tchibo
GOLD
SELECTION
Gazdag íz, intenzív aroma
Instant kávé
100g

From **Figure 7**, three failure cases can be clearly identified:

1. Blurry/low-contrast text (e.g. the text on the porzellan)
2. Custom fonts (e.g. the Tchibo logo)
3. Noisy text that isn't actually meant to be read (e.g. most of the text on the cardboard box)

Reflection

The process used for this project can be summarized using the following steps:

1. An initial problem and relevant, public datasets were found
2. The data was downloaded and preprocessed (segmented)
3. A benchmark was created for the classifier
4. The classifier was trained using the data (multiple times, until a good set of parameters were found)
5. The TensorFlow Android demo was adapted to run the classifier
6. The application was extended so that it can extract text from images using the Google Cloud Vision API
7. Feeding the extracted text to the TTS system was implemented

I found steps 4 and 5 the most difficult, as I had to familiarize myself with the files of the TensorFlow Android demo, which uses Bazel and the Android NDK, both of which were technologies that I was not familiar with before the project.

As for the most interesting aspects of the project, I'm very glad that I found the COCO and COCO-Text datasets, as I'm sure they'll be useful for later projects/experiments. I'm also happy about getting to use TensorFlow, as I believe it will be **the** deep learning library in the future.

Improvement

To achieve the optimal user experience, using more capable hardware[9] and moving the text extraction process from the cloud to the device would be essential. This would reduce the processing time and give access to the outputs of all of the modules of the text extraction pipeline, which would, in turn, enable the following features:

- User-guided reading (e.g. read big text first, or read the text the user is pointing at)
- Better support for languages other than English
- Output filtering (e.g. ignore text smaller than some adjustable threshold)
- Passive text detection (auditory cue on text detection, perhaps with additional information encoded in the tone and volume)

The user experience could also be improved significantly by using MXNet, which is a deep learning library that is better optimized for mobile devices than TensorFlow. The speedup wouldn't be enough for running text extraction on the device, but it would reduce the classification delay significantly.

[9] The Jetson TX1 seems like a good candidate, as it is CUDA-capable, thus supported by virtually all modern deep learning libraries. It also has a small form-factor, so the final hardware setup could be a glass-mounted camera discreetly connected to the main device in the user's pocket.