# Machine Learning Engineer Nanodegree

**Capstone Project**

Ann-Kristin Juschka

May 17, 2019

## I. Definition

### Project Overview

In this Nanodegree, *Convolutional Neural Networks (CNNs)* are introduced that are most commonly used to analyze visual data. In the deep learning project of the Nanodegree, we train CNNs to detect dogs in images and to classify dog breeds. In general, object detection and classification is a classical task in machine learning.

The aim of this project is to use CNNs to detect and classify brands in images with logos. For this, usually CNNs are trained with the dataset *FlickrLogos-27* [KPT$^+$11] and *FlickrLogos-32* [RPLvZ11]; e.g. in [BBMS17, ISGK15]. We are also interested in this task as we later wish to *analyze sentiments* in twitter tweeds with pictures where a given company's logo is detected. Due to time and size limits, we focus in this project on *logo detection and classification.*

### Problem Statement

While most logo datasets like *FlickrLogos-27* [KPT$^+$11] and *FlickrLogos-32* [RPLvZ11] contain raw original logo graphics, we want to train our CNN on the in-the-wild logo dataset *Logos in the Wild* [THMB18] whose images contain logos as natural part. As this dataset includes in total 11,054 images with 32,850 annotated logo bounding boxes of 871 brands, it should be possible to train a CNN that achieves a high accuracy or mean average precision (map). This is a challenging task as the regions containg logos are small.

The main goal of this project is to use CNNs to *classify the brand and company logos* from the Logos in the Wild dataset with high accuracy and mean average precision (map). For this, we first train an own CNN architecture as done in the dog breeds project of the Nanodegree. Furthermore, to improve loss and accuracy we make use of the standard CNN architectures *VGG19, Resnet50, InceptionV3, or Xception* that are already trained on the *ImageNet* database [ker]. Moreover, as the dataset comes with annotations in Pascal-VOC style, if time permits we will also train a *Faster Region-based Convolutional Neural Network (Faster R-CNN)* [RHGS15] for two stages: First, an

*Region Proposal Network (RPN)* for *logo detection*, and second, a classifier like VGG19 for logo classification of the candidate regions. This Faster R-CNN will be evaluated using the *mean average precision (map)* metric.

### Metrics

For purely logo classification, we seek to achieve a high accuracy. *Accuracy* is simply the number of correct predictions divided by the total number of predictions.

For logo detection, we need a different metric: *mean average precision (map)* that is the mean over all classes, of the interpolated *average precision* [EVGW+10] for each class. Recall that

$$\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}, \quad \text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}.$$

Considering the precision-recall curve for a given threshold, and the interpolated precision $\text{precision}_{\text{interpolated}}(\text{recall}_i) = \max_{\tilde{r}, \tilde{r} \geq \text{recall}_i} \text{precision}(\tilde{r})$ of the 11 values $\text{recall}_i = \{0, 0.1, 0.2, \ldots, 0.9, 1.0\}$. Then

$$\text{average precision} = \frac{1}{11} \sum_{1}^{11} \text{precision}_{\text{interpolated}}(\text{recall}_i) \qquad [\text{EVGW}^+10, \S\, 4.2].$$

## II. Analysis

### Data Exploration

To train our model, we want to use the recent logo dataset *Logos in the Wild* [THMB18]. As mentioned above, the lastest version of this dataset (v2.0) contains 11,054 images with 32,850 annotated logo bounding boxes of 871 brands and it is collected by performing Google image searches with well-known brand and company names directly or in combination with a predefined set of search terms like 'advertisement', 'building', 'poster' or 'store'. The logo annotations are in Pascal-VOC style.

As stated by the creators of the Logo in the Wild dataset, this dataset has 4 to 608 images per searched brand, 238 of 871 brands occur at least 10 times, and there are up to 118 logos in one image. Unfortunately, the dataset provides only the links to the images, and some of these images already disappeared. As we later want to detect logos in arbitrary pictures from twitter tweeds, this large in-the-wild logo dataset still fits best to our goal.

Instead of downloading ourselves the images from the different urls provided in the Logos in the Wild dataset, we download the *QMUL-OpenLogo Dataset* [SZG18], which contains Logos in the Wild as a subset including all available JPEG files.

We run a simple Python script to analyze the downloaded JPEG images in the Logos in the Wild dataset, and we find that in fact there are in total 821 brands, where the maximal number of logos per given brand is 1,928 for Heineken, and the minimum number is 1. Moreover, the image heineken/img00042.jpg contains the maximum of 118 logos, while in general every image contains at least one logo.

## Exploratory Visualization


Figure 1: `https://www.iosb.fraunhofer.de/servlet/is/78045/`

While the images on the right contain logos of a single brand, the image on the left shows a sample image that includes bounding boxed for logos of different brands. In particular, an image can contain different types of logos of a brand. The following are the annotations in the Pascal-VOC style in the XML file corresponding to the image on the left hand side:

```
<annotation>
  <folder>0samples</folder>
  <filename>img000009</filename>
  <path>\0samples\img000009.jpg</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>718</width>
    <height>535</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>starbuckscoffee</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>466</xmin>
      <ymin>133</ymin>
      <xmax>709</xmax>
      <ymax>287</ymax>
    </bndbox>
  </object>
  <object>
    <name>starbucks-symbol</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>193</xmin>
      <ymin>270</ymin>
      <xmax>261</xmax>
      <ymax>360</ymax>
    </bndbox>
  </object>
  <object>
    <name>starbucks-symbol</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>420</xmin>
      <ymin>423</ymin>
      <xmax>464</xmax>
      <ymax>513</ymax>
    </bndbox>
  </object>
  <object>
    <name>six</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>633</xmin>
      <ymin>327</ymin>
      <xmax>671</xmax>
      <ymax>365</ymax>
    </bndbox>
  </object>
  <object>
    <name>tchibo</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>8</xmin>
      <ymin>312</ymin>
      <xmax>34</xmax>
      <ymax>351</ymax>
    </bndbox>
  </object>
</annotation>
```

## Algorithms and Techniques

As mentioned before, we start by training an own CNN architecture to classify the logos as done in the dog breeds project of the Nanodegree. For this, 40% of the dataset forms the test set, and the remaining data is split into 80% training set and 20% validation set. Using Keras preprocessing, each image is converted into a 4D tensor with shape $(1, 224, 224, 3)$. This CNN consists of a convolutionary layer, a max-pooling layer, another convolutionary layer, a max-pooling layer, a global-average pooling layer and a final fully connected layer.

The theory behind selecting these layer is that at first nodes in *convolutionary layers* help to detect patterns in single small regions in the image by "filtering" images that are interpreted as 3D arrays (height, width and color). In order to reduce overfitting caused by the high-dimensionality of the filter stack in the convolutionary layers we further use the following two types of *pooling layers*: Nodes in the first type *max-pooling layer* contain the maximum value of the corresponding region in the filter stack of the previous layer. *Global average pooling layers* reduce dimensionality more drastically by taking the average of the entries of each of the three 2D arrays, which constitute the image as 3D array, to produce a 3D vector.

As gradient descent optimizer algorithm we choose "RMSprop", as loss function for our categorization problem "categorical cross entropy" and as metric "accuracy". Using Keras ModelCheckpoint, we save the model with the best validation loss.

As a next step we train a popular CNN like VGG19 whose weights are pre-trained on ImageNet. We proceed similarly as in the first setting.

Having tried CNNs for logo classification, we finally proceed to Faster R-CNNs for logo detection and classification. We first train a *Region Proposal Network (RPN)* that proposes regions with logos in images. The predicted region proposals are then reshaped using a *Region of Interest (RoI)* pooling layer. This layer is next used to classify the image within the proposed region and predict the offset values for the bounding boxes. For the latter task, we again train a CNN like VGG19 that is pre-trained on ImageNet. As explained above, here we use mean average precision as metric.

## Benchmark

The recent *Logos in the Wild* dataset has not been studied much yet. When introduced in [THMB18], the focus is put on open set logo retrieval where only one sample image of a logo is available.

Instead we want to focus on a closed world assumption where we train and test on the Logos in the Wild dataset which has multiple images per brand. Therefore, we can only compare our results with the ones of other models that were trained and tested on the popular closed dataset *FlickrLogos-32* [RPLvZ11]. As cited in [THMB18], the mean average precision (map) in state-of-the-art results is 0.811 achieved by Faster-RCNN [SZG16] where the training set is expanded with synthetically generated training images, and 0.842 using Fast-M [BLF$^+$16] that is a multi-scale Fast R-CNN based-approach.

# III. Methodology

## Data Preprocessing

Having gained access, we download the Logos in the Wild dataset [THMB18] that contains XML files with the bounding boxes, the URLs to the JPEG images, samples and a script to obtain a clean dataset. As stated above, we obtain the JPEG images of the Logos in the Wild dataset by downloading the superset QMUL-OpenLogo Dataset [SZG18].

After setting the variable "oldpath" to the absolute path of our `LogosInTheWild-v2/data` directory and the variable "new path" to the absolute path of our `openlogo/JPEGImages` directoy in our script move_JPEG_images.sh from the Logo Capstone Project, we execute this script that moves the JPEG files from the `openlogo/JPEGImages` directory in the corresponding brand subdirectory in the `LogosInTheWild-v2/data` directory.

Next we remove the `0samples` folder from the `LogosInTheWild-v2/data` directory, and execute in a separate Conda environment with Python 2.7 and opencv-python

```
# From LogosInTheWild-v2/scripts directory
python create_clean_dataset.py --roi --in ../data --out ../cleaned
    -data
```

that adjusts the brand names in the XML files and removes XML files without corresponding JPEG image. This outputs that 9,428 images and 821 brands were processed, while 1,330 JPEG files were unavailable. In addition, it created 28,007 ROI images in corresponding folders with logo names.

For our first logo classification task with a standard CNN architecture in Keras and Python 3.6, we can load the image files with the `load_files` function from `sklearn.datasets` since the Logos in the Wild dataset is organized in directories with corresponding brand names. Of course, some images like the example in Figure 1 contain logos of different brands so labelling them with the brand directory name may decrease accuracy. Furthermore, we use the `image.load_img` and `image.img_to_array` functions from `keras.preprocessing` to convert the JPEG files to Keras' required 4D tensor format with shape $(1, 224, 224, 3)$. As last preprocessing step for the first standard CNN architecture, we rescale the images by dividing every pixel in each image by 255.

Finally, for training a Faster R-CNN with Tensorflow's Object Detection API we need to convert the Logos in the Wild dataset with annotations in Pascal-VOC style to Tensorflow's TFRecord file format. For this, we copy the Python script `create_and_analyze_pascal_tf_record.py` from the Capstone project folder to the LogosInTheWild-v2 directory, rename the folder `LogosInTheWild-v2/cleaned_data` to `LogosInTheWild-v2/data`, enter "export PYTHONPATH=$PYTHONPATH:‘pwd‘:‘pwd‘/slim " when in the `tensorflow/models/research` directory and run the following:

```
# From LogosInTheWild-v2 directory
python create_and_analyze_pascal_tf_record.py --data_dir=./data/
    voc_format --label_map_path=./data/pascal_label_map.pbtxt --
    output_path=./data/
```

After completion we see that we converted 6,034 images with annotations into the 10 training TFRecord files `pascal_train.record-0000i-of-00010`, and 1,508 images with annotations into the 10 validation TFRecord files `pascal_val.record-0000i-of-00010` for $i = 0, \ldots, 9$. Further, in `LogosInTheWild-v2/data` this created `test_images.txt` containing the absolute path of the 1,886 images in the test set, and `pascal_label_map.pbtxt` containing 821 entries like the following:

```
item {
  id: 262
  name: 'starbucks-text'
}
```

Among other parameters for our Faster R-CNN, we choose in our pipeline configuration file `faster_rcnn_inception_logos-locally-on-ubuntu.config` by the Capstone Project as data_augmentation_options random_horizontal_flip, random_vertical_flip and random_rotation90.

## Implementation

### A first CNN model from scratch in Keras [C+15]

For the logo classification task, we start to implement a standard Keras CNN architecture in the Jupyter notebook `logos.ipynb` located in the Capstone Project repository. Keras' model summary is as follows:

Table 1: Architecture of a first CNN for logo classification from Keras.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d_3 (Conv2D) | (None, 223, 223, 32) | 416 |
| max_pooling2d_4 (MaxPooling2) | (None, 111, 111, 32) | 0 |
| conv2d_4 (Conv2D) | (None, 110, 110, 64) | 8256 |
| max_pooling2d_5 (MaxPooling2) | (None, 55, 55, 64) | 0 |
| global_average_pooling2d_2 (GAP2D[1]) | (None, 64) | 0 |
| dense_2 (Dense) | (None, 109) | 7085 |

This model has in total 15,757 parameters that are all trainable. We compile the model with optimizer 'rmsprop' and loss 'categorical_crossentropy', and choose as metric 'accuracy'. For training, we set epochs to 100, validation_split to 0.3 and define a `ModelCheckpoint` from `keras.callbacks` that saves the model with the best validation loss.

After training and loading the model with the best weights, we compute the accuracy on the test images:

```
print('\n', 'Test accuracy of the model from scratch:',
    model_from_scratch.evaluate(test_tensors, test_targets, verbose
    =0)[1]*100,'%.')
```

[1] GlobalAveragePooling2D

After training for 20 epochs, this results in "Test accuracy of the model from scratch: 16.76 %". Finally, continuing to train for 600 further epochs leads to a test accuracy of 33.99 % where the validation loss improved for the last time in epoch 594 to 2.95990.

In the next section, we describe how we fine-tune our first model.

### A Custom Model on top of the VGG16 model [SZ15] in Keras

Now we build a custom Keras CNN model on top of the well-known VGG16 architecture:

```
from keras.applications.vgg16 import VGG16
VGG16_model = VGG16(include_top=True, weights='imagenet',
    input_shape=train_tensors.shape[1:])
```

Let us first summarize the VGG16 model:

Table 2: Architecture of the VGG16 model from Keras.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_2 (InputLayer) | (None, 224, 224, 3) | 0 |
| block1_conv1 (Conv2D) | (None, 224, 224, 64) | 1792 |
| block1_conv2 (Conv2D) | (None, 224, 224, 64) | 36928 |
| block1_pool (MaxPooling2D) | (None, 112, 112, 64) | 0 |
| block2_conv1 (Conv2D) | (None, 112, 112, 128) | 73856 |
| block2_conv2 (Conv2D) | (None, 112, 112, 128) | 147584 |
| block2_pool (MaxPooling2D) | (None, 56, 56, 128) | 0 |
| block3_conv1 (Conv2D) | (None, 56, 56, 256) | 295168 |
| block3_conv2 (Conv2D) | (None, 56, 56, 256) | 590080 |
| block3_conv3 (Conv2D) | (None, 56, 56, 256) | 590080 |
| block3_pool (MaxPooling2D) | (None, 28, 28, 256) | 0 |
| block4_conv1 (Conv2D) | (None, 28, 28, 512) | 1180160 |
| block4_conv2 (Conv2D) | (None, 28, 28, 512) | 2359808 |
| block4_conv3 (Conv2D) | (None, 28, 28, 512) | 2359808 |
| block4_pool (MaxPooling2D) | (None, 14, 14, 512) | 0 |
| block5_conv1 (Conv2D) | (None, 14, 14, 512) | 2359808 |
| block5_conv2 (Conv2D) | (None, 14, 14, 512) | 2359808 |
| block5_conv3 (Conv2D) | (None, 14, 14, 512) | 2359808 |
| block5_pool (MaxPooling2D) | (None, 7, 7, 512) | 0 |
| flatten (Flatten) | (None, 25088) | 0 |
| fc1 (Dense) | (None, 4096) | 102764544 |
| fc2 (Dense) | (None, 4096) | 16781312 |
| predictions (Dense) | (None, 1000) | 4097000 |

This model has a large number of 138,357,544 parameters that are all trainable.

On top of this VGG16 model, we add some custom layers as follows:

Table 3: Our custom Architecture on top of the above VGG16 model.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_2 (InputLayer) | (None, 224, 224, 3) | 0 |
| block1_conv1 (Conv2D) | (None, 224, 224, 64) | 1792 |
| block1_conv2 (Conv2D) | (None, 224, 224, 64) | 36928 |
| block1_pool (MaxPooling2D) | (None, 112, 112, 64) | 0 |
| block2_conv1 (Conv2D) | (None, 112, 112, 128) | 73856 |
| block2_conv2 (Conv2D) | (None, 112, 112, 128) | 147584 |
| block2_pool (MaxPooling2D) | (None, 56, 56, 128) | 0 |
| block3_conv1 (Conv2D) | (None, 56, 56, 256) | 295168 |
| block3_conv2 (Conv2D) | (None, 56, 56, 256) | 590080 |
| block3_conv3 (Conv2D) | (None, 56, 56, 256) | 590080 |
| block3_pool (MaxPooling2D) | (None, 28, 28, 256) | 0 |
| block4_conv1 (Conv2D) | (None, 28, 28, 512) | 1180160 |
| block4_conv2 (Conv2D) | (None, 28, 28, 512) | 2359808 |
| block4_conv3 (Conv2D) | (None, 28, 28, 512) | 2359808 |
| block4_pool (MaxPooling2D) | (None, 14, 14, 512) | 0 |
| block5_conv1 (Conv2D) | (None, 14, 14, 512) | 2359808 |
| block5_conv2 (Conv2D) | (None, 14, 14, 512) | 2359808 |
| block5_conv3 (Conv2D) | (None, 14, 14, 512) | 2359808 |
| block5_pool (MaxPooling2D) | (None, 7, 7, 512) | 0 |
| conv2d_3 (Conv2D) | (None, 5, 5, 64) | 294976 |
| max_pooling2d_4 (MaxPooling2D) | (None, 2, 2, 64) | 0 |
| global_average_pooling2d_2 (GAP2D[2]) | (None, 64) | 0 |
| dropout_1 (Dropout) | (None, 64) | 0 |
| dense_2 (Dense) | (None, 109) | 7085 |

Since we only set our custom layers as trainable, out of the 15,016,749 total parameters, only 302,061 are trainable. After training for 20 epochs, this results in a very good test accuracy of 36.80 %". However, when continuing to train the custom model for 600 further epochs, the training loss kept decreasing but the validation loss increased. This indicates overfitting, and since after 20 epochs still no weights with an improved validation loss were achieved, we then stopped the training.

### A custom Faster R-CNN with Tensorflow's Object Detection API [HRS+16]

Finally, we want to improve our model by using Tensorflow's Object Detection API. After the necessary preprocessing steps described in section III, we define our Faster R-CNN with the following configuration file `faster_rcnn_inception_logos-locally-on-ubuntu.config` that is located in the Logo Capstone Repository.

[2]GlobalAveragePooling2D

```
model {
  faster_rcnn {
    num_classes: 821
    image_resizer {
      keep_aspect_ratio_resizer {
        min_dimension: 600
        max_dimension: 1024
      }
    }
    feature_extractor {
      type: 'faster_rcnn_inception_v2'
      first_stage_features_stride: 16
    }
    first_stage_anchor_generator {
      grid_anchor_generator {
        scales: [0.25, 0.5, 1.0, 2.0]
        aspect_ratios: [0.5, 1.0, 2.0]
        height_stride: 16
        width_stride: 16
      }
    }

    first_stage_box_predictor_conv_hyperparams {
      op: CONV
      regularizer {
        l2_regularizer {
          weight: 0.0
        }
      }
      initializer {
        truncated_normal_initializer {
          stddev: 0.01
        }
      }
    }
    first_stage_nms_score_threshold: 0.0
    first_stage_nms_iou_threshold: 0.7
    first_stage_max_proposals: 300
    first_stage_localization_loss_weight: 2.0
    first_stage_objectness_loss_weight: 1.0
    initial_crop_size: 14
    maxpool_kernel_size: 2
    maxpool_stride: 2
    second_stage_box_predictor {
      mask_rcnn_box_predictor {
        use_dropout: false
        dropout_keep_probability: 1.0
        fc_hyperparams {
          op: FC
          regularizer {
            l2_regularizer {
              weight: 0.0
            }
          }
          initializer {
            variance_scaling_initializer {
              factor: 1.0
              uniform: true
              mode: FAN_AVG
            }
          }
        }
      }
    }
    second_stage_post_processing {
      batch_non_max_suppression {
        score_threshold: 0.0
        iou_threshold: 0.6
        max_detections_per_class: 100
        max_total_detections: 300
      }
      score_converter: SOFTMAX
    }
    second_stage_localization_loss_weight: 2.0

    second_stage_classification_loss_weight: 1.0
  }
}
train_config: {
  batch_size: 1
  optimizer {
    momentum_optimizer: {
      learning_rate: {
        manual_step_learning_rate {
          initial_learning_rate: 0.0002
          schedule {
            step: 900000
            learning_rate: .00002
          }
          schedule {
            step: 1200000
            learning_rate: .000002
          }
        }
      }
      momentum_optimizer_value: 0.9
    }
    use_moving_average: false
  }
  gradient_clipping_by_norm: 10.0
  num_steps: 5000

  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
    random_vertical_flip {
    }
  }
  data_augmentation_options {
    random_rotation90 {
    }
  }
}

train_input_reader {
  label_map_path: "PATH_TO/LogosInTheWild-v2/data/pascal_label_map.pbtxt"
  tf_record_input_reader {
    input_path:"PATH_TO/LogosInTheWild-v2/data/pascal_train.record-?????-of-00010"
  }
}
eval_config {
  num_examples: 1886
  max_evals: 1886
  #use_moving_averages: false
  metrics_set: "pascal_voc_detection_metrics"
}
eval_input_reader {
  label_map_path:  "PATH_TO/LogosInTheWild-v2/data/pascal_label_map.pbtxt"
  shuffle: false
  num_readers: 10
  tf_record_input_reader {
    input_path: "PATH_TO/LogosInTheWild-v2/data/pascal_val.record-?????-of-00010"
  }
}
```

We make sure that the directory structure is as recommended in running_locally.md:

```
+ "data" directory
  - pascal_label_map.pbtxt
  - 10 files pascal_train.record-0000i-of-00010 for i = 0,...,9
  - 10 files pascal_val.record-0000i-of-00010 i = 0,...,9
+ "models" directory
  + "model" directory
    - faster_rcnn_inception_logos-locally-on-ubuntu.config
    + "train" directory
    + "eval" directory,
```

and train our model by running the following Python script:

```
# From tensorflow/models/research directory
export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim
python object_detection/model_main.py \
    --model_dir=PATH_TO/LogosInTheWild-v2/models/model/ \
    --pipeline_config_path=PATH_TO/LogosInTheWild-v2/models/model/
   faster_rcnn_inception_logos-locally-on-ubuntu.config \
    --num_train_steps=50000 --alsologtostderr
```

We monitor statistics with

```
tensorboard --logdir=PATH_TO/LogosInTheWild-v2/models/model/.
```

After training finishes, we can export our model checkpoint:

```
# From tensorflow/models/research/ directory
CHECKPOINT_NUMBER= Number from "model.ckpt-${CHECKPOINT_NUMBER}.
   meta"
python object_detection/export_inference_graph.py \
    --input_type=image_tensor \
    --pipeline_config_path=PATH_TO/LogosInTheWild-v2/models/model/
   faster_rcnn_inception_logos-locally-on-ubuntu.config \
    --trained_checkpoint_prefix=PATH_TO/LogosInTheWild-v2/models/
   model/model.ckpt-${CHECKPOINT_NUMBER} \
    --output_directory=PATH_TO/LogosInTheWild-v2/export
```

By opening `LogoCapstoneProject/adjusted_object_detection_tutorial.ipynb` in a jupyter notebook, adjusting the paths and executing the cells we can explore the results on the sample images.

Unfortunately, neither when evaluating nor when making predictions with the frozen model any logos are detected (see `https://github.com/tensorflow/models/issues/6748`). To overcome this problem, we trained with only four logo classes instead of the total 821 classes but still no objects were detected. We also tried different conda environments: with Python 2.7, 3.6 or 3.7, on Windows 10, Ubuntu 18.04, Ubuntu 18.10 and even training on Google Cloud as described here. Also using `tensorflow/models/research/object_detection/legacy/train.py` and `eval.py` instead of `model_main.py` for training and evaluating did not lead to any detections. At first, we used a

label map that was created by scratch but the detections did not improve with an automatically created label map. Furthermore, different pipeline configs did not lead to object detections either.

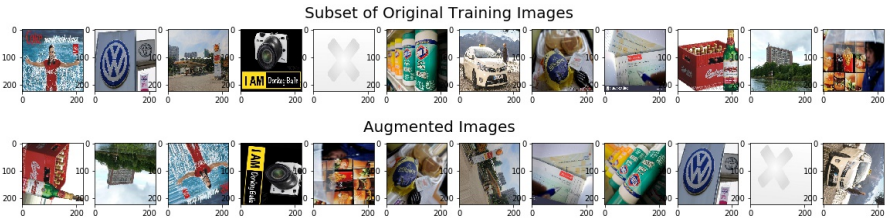
### Refinement

As a first step for all our Keras models, we use Keras Data Augmentation to augment the training data. This is important as many logo classes do only contain few images.

```
from keras.preprocessing.image import ImageDataGenerator
datagen_train = ImageDataGenerator(
    width_shift_range=0.1,  # randomly shift images horizontally (10% of total width)
    height_shift_range=0.1,  # randomly shift images vertically (10% of total height)
    horizontal_flip=True, # randomly flip images horizontally
    vertical_flip=True, # randomly flip images vertically
    rotation_range=90) #randomly rotate images by up to 90 degrees
datagen_valid = ImageDataGenerator(
    width_shift_range=0.1,  # randomly shift images horizontally (10% of total width)
    height_shift_range=0.1,  # randomly shift images vertically (10% of total height)
    horizontal_flip=True, # randomly flip images horizontally
    vertical_flip=True, # randomly flip images vertically
    rotation_range=90) #randomly rotate images by up to 90 degrees

datagen_train.fit(train_tensors)
datagen_valid.fit(val_tensors)
```

We visualize how one augmented image looks for 12 images from the training set:




At first, we state that the test accuracy is 28.79 % after continuing to train our model from scratch from Table 1 on the training set with augmented images for 50 epochs. The validation accuracy at epoch 50 is slightly higher than the one of epoch 50 when we trained only on the original images – but of course the results are not really comparable as we started with pre-trained weights.

Similarly, we report that the test accuracy is 28.10 % after continuing to train our custom model from Table 3 on the training set with augmented images for 20 epochs. However, here we actually stopped training after 20 epochs as accuracy on the training set stays constantly around 17 % and neither does training loss improve.

Since without augmented images the custom model overfits and with augmented images does not learn, we decide to finetune the model from Table 1.

To achieve a higher complexity, we add further layers to this model as follows:

Table 4: Architecture of an extended CNN from scratch.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d_1_input (InputLayer) | (None, 224, 224, 3) | 0 |

*Continued on next page*

Table 4 – *Continued from previous page*

| | | |
|---|---|---|
| conv2d_1 (Conv2D) | (None, 223, 223, 32) | 416 |
| max_pooling2d_1 (MaxPooling2) | (None, 111, 111, 32) | 0 |
| conv2d_2 (Conv2D) | (None, 110, 110, 64) | 8256 |
| max_pooling2d_2 (MaxPooling2) | (None, 55, 55, 64) | 0 |
| conv2d_4 (Conv2D) | (None, 53, 53, 32) | 18464 |
| max_pooling2d_4 (MaxPooling2) | (None, 26, 26, 32) | 0 |
| conv2d_5 (Conv2D) | (None, 25, 25, 64) | 8256 |
| max_pooling2d_5 (MaxPooling2) | (None, 8, 8, 64) | 0 |
| dropout_2 (Dropout) | (None, 8, 8, 64) | 0 |
| conv2d_6 (Conv2D) | (None, 7, 7, 32) | 8224 |
| max_pooling2d_6 (MaxPooling2) | (None, 3, 3, 32) | 0 |
| conv2d_7 (Conv2D) | (None, 2, 2, 64) | 8256 |
| global_average_pooling2d_3 (GAP2D[3]) | (None, 64) | 0 |
| dropout_3 (Dropout) | (None, 64) | 0 |
| dense_3 (Dense) | (None, 109) | 7085 |

This model has 58,957 parameters that are all trainable.

After training for 50 epochs, this extended model reaches an accuracy of 26.03 % on the test images.

Since Scikit-learn's GridSearchCV hangs after the fourth step when trying to fine-tune the parameters optimizer, activation function and batch size, we test some parameter combinations by hand.

We start by choosing optimizer "Adam" instead of "RMSprop", and a batch size of 1 instead of 32. The following epochs both training and validation loss increases while training and validation accuracy decreases. We note that this is similar to what happens when we train the custom model with the augmented images.

In the case of the custom model, we increase the batch size to 128, keep optimizer "RMSprop" and continue to train for 20 further epochs, which leads to an accuracy of 30.91 % on the test set.

In the case of the extended model from scratch, we increase batch size to 64, keep otimizer "Adam" and continue to train for 30 further epochs, which leads to an accuracy of 29.85 % on the test set.

Next, for the extendend model from scratch we choose a batch size to 64, otimizer "adagrad" and continue to train for 30 further epochs, which leads to an accuracy of 29.80 % on the test set. Finally, after 100 epochs we achieve a test accuracy of 30.91 %.

Summing up, the test accuracies of the models are still very similar and the best accuracy seems to be reached when training for a longer time.

Since the results are very similar, for theoretical reasons we decide to choose the extended model from scratch with optimizer Adam. More precisely, the authers [KB15] designed "Adam" to combine the advantages of two optimizers: AdaGrad [DHS11] that works well with sparse gradients, and RMSProp [TH12]. However, when we continue to

[3]GlobalAveragePooling2D

train this model for further 600 epochs, the weights with the best validation loss do not get updated after epoch 138 anymore so we interrupt training after 408 epochs. Loading the best weights from epoch 138 results in a test accuracy of 31.18 %.

# IV. Results

## Model Evaluation and Validation

As most models performed similarly, we choose the extended model from scratch from Table 4 as its architecture has a medium level of complexity, and the extended model did not overfit like the custom model on top of the VGG16 model.

Trained on the augmented images, the extended model from scratch shows litte sensitivity to changing the optimizer and batch size. From the training statistics, we get the impression that training it for more epochs would not lead to better results.

```
In [61]: # plot the training loss and accuracy
         plt.style.use("ggplot")
         plt.figure()
         N = 10
         plt.plot(np.arange(0, N), H63.history["loss"], label="train_loss")
         plt.plot(np.arange(0, N), H63.history["val_loss"], label="val_loss")
         plt.plot(np.arange(0, N), H63.history["acc"], label="train_acc")
         plt.plot(np.arange(0, N), H63.history["val_acc"], label="val_acc")
         plt.title("Training Loss and Accuracy")
         plt.xlabel("Epoch #")
         plt.ylabel("Loss/Accuracy")
         plt.legend(loc="upper left")

Out[61]: <matplotlib.legend.Legend at 0x7f262ab6ed30>
```


## Justification

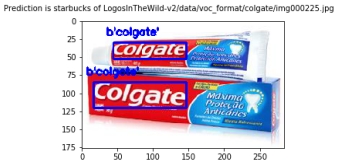
Unfortunately, we did not see other research results for logo classification with metric accuracy to compare these to our results. However, given that the dataset had many logo classes with only few images, we think that we cannot expect a high overall accuracy. Still, we would like to reach an accuracy of at least 50 % but we do not know which parameters of the different models from the previous sections, or which other CNN architectures could lead to a test accuracy higher than 40 %.

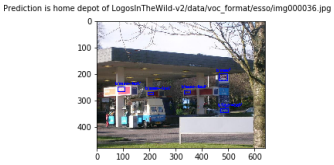
On the other hand, our final model is robust and classifying small logos in larger images is not an easy task so our final result with a test accuracy of 30.91 % is adequate.

# V. Conclusion

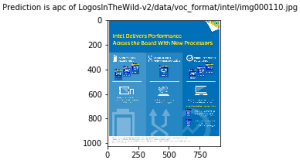
## Free-Form Visualization

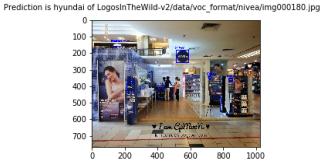
In this subsection, we visualize some logo classification results on ten test images:

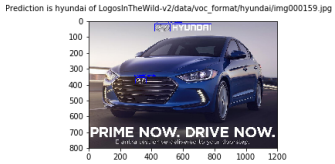


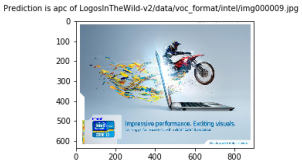


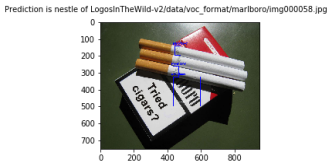







We note that some logo classes like Volkswagen seem to have a better accuracy than the others.

## Reflection

There were several obstacles when training a CNN to classify or detect logos. At first, the local hardware was not sufficient so often the training process freezes or progresses only slowly. Also uploading the large dataset to Amazon Web Services or Google Cloud's bucket took very long. Furthermore, when training a Faster R-CNN with Tensorflow's Object Detection API the error log was not sufficiently detailed to identify why there are no detections.

In addition to these technical limitations, further experience with CNNs is needed for adjusting the right parameters of the models. In the end, the very first two model

architectures from Table 1 and Table 3 already reach the best accuracy.

### Improvement

As a first crucial improvement, one could use Scikit-learn's from MultiLabelBinarizer to label the images with potentially multiple logo classes from the corresponding XML files. This would lead to 821 classes instead of 109 but the logo images may be more accurately classified. How to train a CNN with multiple label classification is described here.

The accuracy may also be improved by (additionaly / pre-)training with the ROI images of the Logos in the Wild dataset, which are generated by the Python script `create\_clean\_dataset.py`. We did not choose this option as our goal is to classify logos in any image. By construction, CNNs are relatively translation invariant so that our goal seems to be within reach.

Moreover, due to time limits we did not train our models very long. More epochs could also lead to higher accuracy and to choosing a different model than we did.

Similarly, there are many more reasonable parameter combinations to explore than we did in this short time. For instance, using different activation functions, various filter sizes or decreasing learning rates.

Last, of course training an object detection model may lead to better results like the one mentioned in our benchmark section. We do not know why we could not detect any logos with our Faster R-CNN but it makes sense to still further investigate this.

## References


[BBMS17] Simone Bianco, Marco Buzzelli, Davide Mazzini, and Raimondo Schettini. Deep learning for logo recognition. *Neurocomputing*, 245:23 – 30, 2017.

[BL15] Yoshua Bengio and Yann LeCun, editors. *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[BLF+16] Yu Bao, Haojie Li, Xin Fan, Risheng Liu, and Qi Jia. Region-based cnn for logo detection. In *Proceedings of the International Conference on Internet Multimedia Computing and Service*, ICIMCS'16, pages 319–322, New York, NY, USA, 2016. ACM.

[C+15] François Chollet et al. Keras. `https://keras.io`, 2015.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[EVGW+10] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[HRS+16] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. *CoRR*, abs/1611.10012, 2016.

[ISGK15] Forrest N. Iandola, Anting Shen, Peter Gao, and Kurt Keutzer. Deeplogo: Hitting logo recognition with the deep neural network hammer. *CoRR*, abs/1510.02131, 2015.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Bengio and LeCun [BL15].

[ker] Keras Documentation. `https://keras.io/applications/`. Online; accessed 2019-03-17.

[KPT+11] Y. Kalantidis, LG. Pueyo, M. Trevisiol, R. van Zwol, and Y. Avrithis. Scalable triangulation-based logo recognition. In *in Proceedings of ACM International Conference on Multimedia Retrieval (ICMR 2011)*, Trento, Italy, April 2011.

[RHGS15] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[RPLvZ11] Stefan Romberg, Lluis Garcia Pueyo, Rainer Lienhart, and Roelof van Zwol. Scalable logo recognition in real-world images. In *Proceedings of the 1st ACM International Conference on Multimedia Retrieval*, ICMR '11, pages 25:1–25:8, New York, NY, USA, 2011. ACM.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Bengio and LeCun [BL15].

[SZG16] Hang Su, Xiatian Zhu, and Shaogang Gong. Deep learning logo detection with data expansion by synthesising context. *CoRR*, abs/1612.09322, 2016.

[SZG18] Hang Su, Xiatian Zhu, and Shaogang Gong. Open logo detection challenge. *CoRR*, abs/1807.01964, 2018.

[TH12] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. *University of Toronto, Technical Report*, 2012.

[THMB18] Andras Tüzkö, Christian Herrmann, Daniel Manger, and Jürgen Beyerer. Open Set Logo Detection and Retrieval. In *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications: VISAPP*, 2018.