# Implementation

## Sprints

The implementation chapter is built on the basic development of the model discussed in the prototype chapter. The initial plan involves six key phases in the development process. Every state includes further divisions (evaluated in the corresponding sprint sections). These phases are as follows:

1. Pre-processing - Prepare labelled images. Split dataset into training and test sample files.
2. Generate TFRecord formatted data from training and test sets.
3. Setup configuration file
4. Train the model (graph)
5. Export trained graph (freeze it)
6. Apply the “frozen” graph to classify logos.

### Sprint 1 – Data preparation

|  |  |  |  |
| --- | --- | --- | --- |
| Sprint Number | Sprint Name | Start Date | Finish Date |
| 1 | Input data preparation | 20/01/2018 | 15/02/2018 |

|  |  |  |
| --- | --- | --- |
| Task Number | Details | Status |
| 1 | Examine the dataset – Flickr27 -used for the application. Differentiate between the provided data and Tensorflow’s feature requirements. | Complete |
| 2 | Convert both test and training files to csv format. | Complete |
| 3 | Add image sizes to existing data. | Complete |
| 4 | Remove unnecessary column from initial data file. | Complete |
| 5 | Create newly structured csv files for conversion. | Complete |

Based on the detailed research undertaken the decision was made to use Tensorflow Object Detection API. This approach is called Transfer Learning. This technique allows skipping working with the millions of parameters that could take weeks to train. Instead, use a model that already has its convolutional neural network weights learned on a pre-defined object recognition tasks. (Jeff Donahue, 2013)

**Task 1**

The official documentation recommends that all data fed into this model must be in TFRecord format. TFRecord is a record-oriented binary format that most Tensorflow applications use. It processed by Tensorflow in a sequential order step by step. Reading data into a Tensorflow program may be achieved with typically three different approaches.

* Feeding – Feeding Python code directly into the Tensor which means directly into the graph
* Preloaded data – Small datasets can be fed as a constant or variable directly in the graph
* Reading from files – An input pipeline reads data at the beginning of the graph. This approach is being demonstrated in the project.

The Flickr dataset’s structure composed of the following elements:

Flickr\_logos\_27\_dataset\_images.tar 🡪 all images in .jpg format, not separated in training and test folders

Flickr\_logos\_27\_dataset\_query\_set\_annotation.txt 🡪 test directory containing: filename and corresponding class name

Flickr\_logos\_27\_dataset\_training\_set\_annotation.txt 🡪 training directory containing: filename, class name, a subclass of the class, xmin, ymin, xmax, ymax (location coordinates)

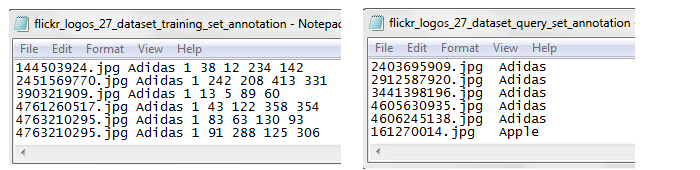


Figure 6.1 – Annotations of the initial dataset

Machine learning tasks such as image processing may be solved either supervised or unsupervised. This project applies the supervised learning approach (see explanation under section 3.4). The technique requires a set of data that is ideally split up into training and testing sets. Flickr’s logo dataset is coming with these pre-organised sections.

**Task 2.**

A simple conversion from txt to csv file formats was required. Reason: this is Tensorflow’s recommended approach. Regardless the level of difficulty some issues already arose at early stages. Discussed further in the Problems section.

**Task 3-5.**

Figure 6.2 (below) illustrates, the process of adding the necessary sizes of the images to the training and testing data. The initial training set had to be truncated first, then the new measurements were added. Also had to make sure, that the file did not contain any replications as it was in the original version. These tasks were achieved by writing several helper csv files until the data reached its final desired format; im\_filename, width, height, classname, xmin, ymin, xmax, ymax

After the merge - between the full set of measurements and the test/training sets - were complete, the missing coordinates in the test file were added manually. Alternatively, third-party software can be used to crop the images, which allows for automated data generation, but given the small size of the testing set, this approach was not necessary.

**Problems**

1. One of the issues encountered was when converting txt files to csv., even though initially the two text files (see Figure 6.1) were to be converted into csv with the same code base, their outcome was slightly different. The test file’s values were not comma separated. This problem was not apparent until later stages when TFRecord was generated and the .record file turned out to be empty. Naturally, this file halted progression with the training stage as well.

The solution to this issue was to use a different approach with the test file. While the training is following a more traditional method, the test text was converted by adding the additional commas to the features individually, closing rows with the last character removal.

1. Another issue was the confusing documentation on TFRecord. This, however, was not an apparent problem until later stages when training was again held up. Conversion into this protobuf was achieved several times yet was forced to return to this stage on multiple occasion as the training was not able to further progress. More on this issue will be covered in Sprint 2.

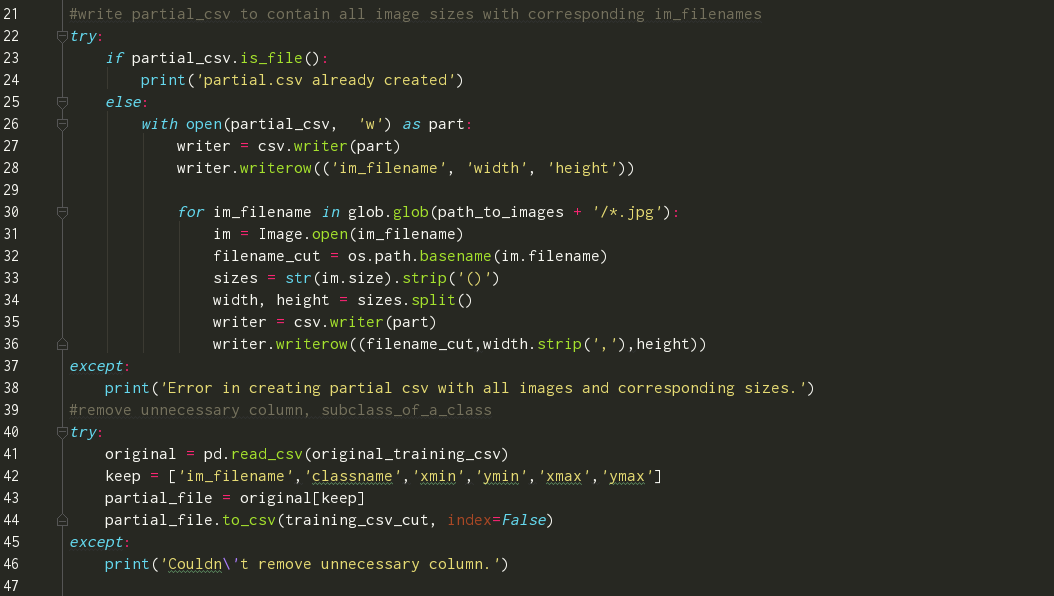


Figure 6.2 – The full process of creating a csv input data file.

### Sprint 2 – Converting data to TFRecord

|  |  |  |  |
| --- | --- | --- | --- |
| Sprint Number | Sprint Name | Start Date | Finish Date |
| 2 | Converting data into TFRecord | 15/02/2018 | 22/03/2018 |

|  |  |  |
| --- | --- | --- |
| Task Number | Details | Status |
| 1 | Tensorflow provides templates for the most common type of dataset conversions.  Analyse the templates. | Complete |
| 2 | Choose a template and tailor it to the flickr dataset. | Complete |
| 3 | Create label map. | Complete |
| 4 | Convert testing and training csv sets into TFRecord. | Complete |

**Task 1**

As indicated in the previous section, Tensorflow documentations in some areas may be a little deceptive. Tensorflow states that a data file, that is to be converted, must be in the following format; *feature 1, feature 2, …, feature n*

This approach is correct of course but written in general terms applying to all(!) the training models that may be created in Tensorflow. This means that the initial data (training set for instance contains: filename, class name, subclass of the class, xmin, ymin, xmax, ymax) should satisfy the requirements. The models from the Object Detection API however, require a more stringent architecture. The first attempts to convert the csv files were based on the original data structure, providing that the test set included no bounding box coordinates, and the training set included an extra, unnecessary column (as indicated earlier). To some extent, the conversion was successful and RECORD file could be generated. The issue again only became apparent at later stages when the API components were unable to work together with the provided files. The training set was perfectly converted given that Tensorflow simply ignored the extra column. The test RECORD, however, was damaged due to the missing elements. All of this has led to aligning the dataset exactly to the expected input requirements.

**Task 2-3**

Once the correct content was set, the code to generate the TFRecord was needed. There are several methods to create these files. The model provides two pre-made scripts for the most typical data structures, a PASCAL VOC data type conversion script and another for the Oxford Pet type of datasets. If none of these is the right fit, a custom code must be written. What is common in all three options are the following;

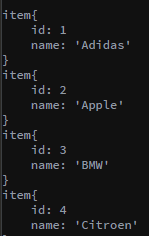
* Input Images must be in either JPEG or PNG format

Figure 2 – Snippet from the label map created

* Input data must include the bounding box coordinates for the location of the desired objects within the image
* Name of the class in the bounding box (encoded utf8)
* A label map must be associated with the dataset. It is a simple JSON-like list, providing a class and an ID for the class. This file must be saved in a protobuf text format (pbtxt)

For this project, a cloned script was used where only a few modifications were needed. It is largely based on a helper code provided by Tensorflow, where the required input data is more generic and better suited for this project than the other two options.

**Task 4**

After running this code (generate\_tfrecord.py) several issues were encountered. This was mainly due to the issues explained in section Sprint 2.1, where the input data was falsely defined. When these problems were eventually corrected the development could proceed to the next stage.

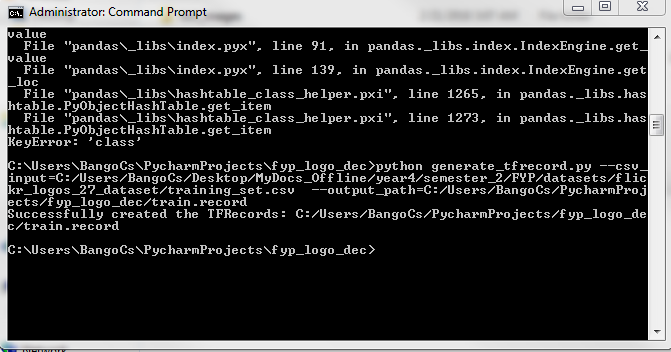


Figure 3 - TFRecord file is generated the first time.

### Sprint 3 – Setup Configuration file

|  |  |  |  |
| --- | --- | --- | --- |
| Sprint Number | Sprint Name | Start Date | Finish Date |
| 3 | Setup configuration file | 22/02/2018 | 01/03/2018 |

|  |  |  |
| --- | --- | --- |
| Task Number | Details | Status |
| 1 | Examine pre-trained models provided by the API | Complete |
| 2 | Define inputs | Complete |
| 3 | Configure the trainer with the right inputs. | Complete |
| 4 | Navigate all files in to the API’s object\_detection folder. | Complete |

**Task 1**

Tensorflow provides several models pre-trained on large datasets. Since the aim of the project is to detect logos on real-time camera footage, a fast model had to be selected. After evaluating the collection of models, the decision was made to use Google’s SSD Mobilenet V1. This model is built on the Single Shot MultiBox Detector (SSD) method. Mobilenet is a feature extension head, that is a series of convolutional blocks from other models such as VGG, Inception etc. It is a neural network, that is specifically designed to run on mobile devices. Its main characteristics are as follows:

* Works efficiently with PASCAL formatted datasets
* Smaller than models such as CNN or the faster RCNN
* Fast – speed: 30 ms

**Task 2-3**

These configuration files support the Transfer Learning objectives. The inputs to be defined are: fine tune checkpoint 🡪 must point to the default model checkpoint, label map path 🡪 points to the label map defined in Sprint 2.3, number of classes 🡪 number corresponding to the logo dataset, input path for both TFRecord files.

**Task 4**

The final step before the training may begin, is to move all the files created into the API’s cloned directory. The training of Tensorflow is initialised by establishing the configurations in the transferred files and the connection between the frozen model and the new input data.

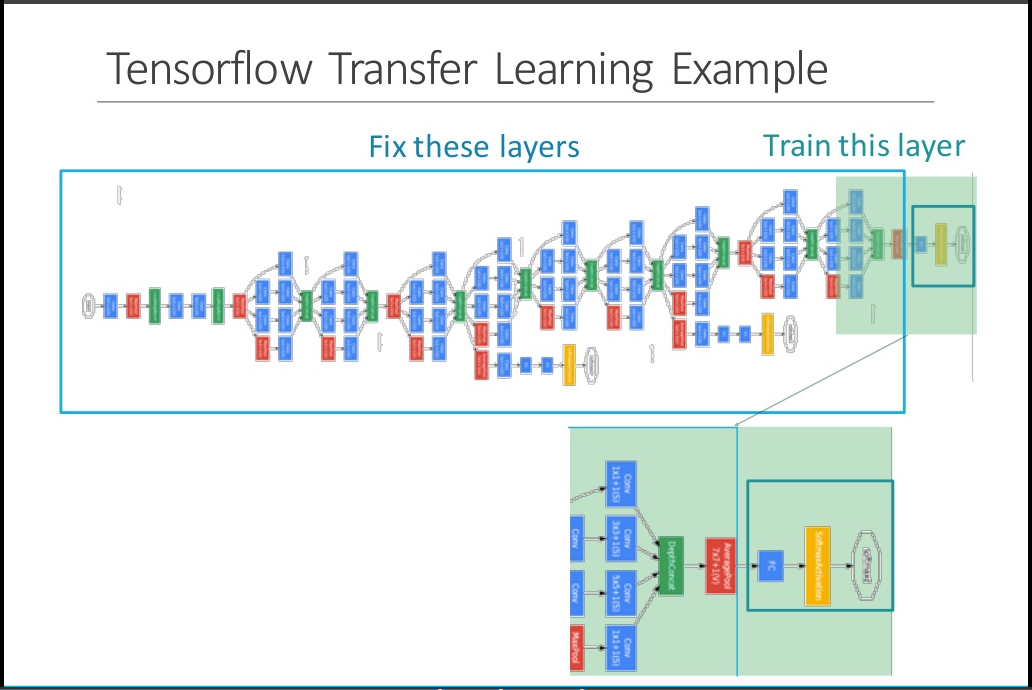


Figure 6.1 – An illustration of Transfer Learning with Tensorflow (Chang, 2016). It is also an accurate visualisation on what was achieved with the configuration file that allowed the training to begin.

### Sprint 4 – Train the model (graph)

|  |  |  |  |
| --- | --- | --- | --- |
| Sprint Number | Sprint Name | Start Date | Finish Date |
| 4 | Training the model (locally) | 01/03/2018 | 15/03/2018 |

|  |  |  |
| --- | --- | --- |
| Task Number | Details | Status |
| 1 | Once the model is chosen, all the configuration jobs are done and the data is in the right format, the training may begin. Usually, with CPU-only Tensorflow setup, this training takes hours.  When Tensorflow starts its training, the following tasks are executed:   * Read in provided files * Create new empty config files where the process gets saved step by step * Iterate through the data until the model is not overfitted. * Produce visualisation graphs on TensorBoard | Complete |

**Task 1**

By summarising the previous steps, the following prerequisites must be fulfilled:

* Training and testing data must be in the right format and are converted into record files
* The configuration file is set
* Label map is created
* All the above along with a folder of images are injected in the API

To train the model the API’s train.py script must be called initialising the training directory and the pipeline config path. The training directory contains the config file and the labelmap. Once the training has begun, all ongoing information (checkpoints, Tensorboard data etc.) are directed here. The pipeline config file points to the config file that has been modified in Sprint 3.

**Problems**:

There are a number of issues that may prevent the training process from being executed. The Object Detection API uses Protobufs to configure the training parameters. „Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data” (Google Developers Official, 2018). A very typical issue (with Windows OS) occurs at the compilation of these proto files. According to Tensorflow’s documentation, the following command should execute this process;

protoc object\_detection/protos/\*.proto --python\_out=.

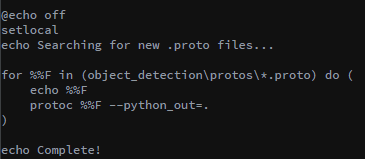
However, Windows has a problem passing in multiple files into proto execution. Solve this problem by adding an a .bat script (Figure 7.1.) (Anon., 2017). Once this file is added the slim directories can be appended to PYTHONPATH (See Figure 7.2).

Figure 7.1

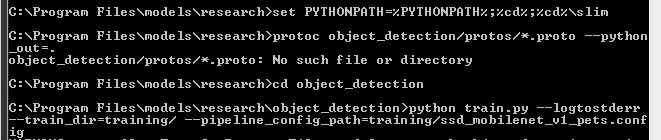
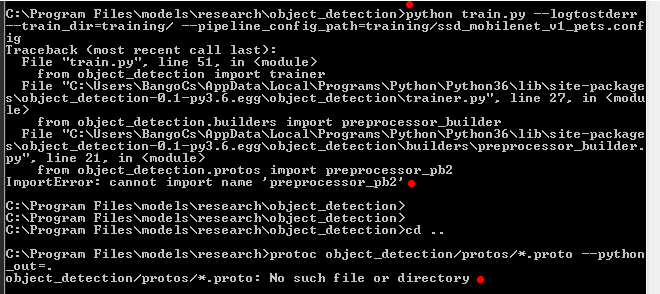


Figure 7.2

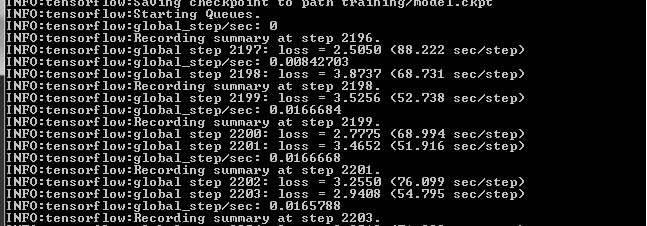


Figure 7.3

### Sprint 5 – Export trained model

|  |  |  |  |
| --- | --- | --- | --- |
| Sprint Number | Sprint Name | Start Date | Finish Date |
| 5 | Export inference graph | 22/03/2018 | 15/03/2018 |

|  |  |  |
| --- | --- | --- |
| Task Number | Details | Status |
| 1 | The model is trained and it’s training steps are saved along with it. However, this model is not reusable just yet. ‘Freezing’ the model is essential for the next phases. | Complete |

Figure 7.3 illustrates the process of the training. All these steps along with the periodically saved checkpoints are saved to a pre-defined path. As for many general tasks, Tensorflow provides a general script as support. export\_inference\_graph.py can be considered as another configuration file that must be tailored to the specific task. Input type, config file path, the last trained checkpoint and a name for the output directory must be specified as arguments. The aim is to ‘freeze’ the inference graph.

The necessary information can be provided when calling the execution;

python export\_inference\_graph.py

--input\_type image\_tensor

--pipeline\_config\_path training/ssd\_mobilenet\_v1\_pets.config

--trained\_checkpoint\_prefix training/model.ckpt-2615

--output\_directory logo2615\_graph

The first input to modify is the pipeline\_config\_file. This is the config file that was initialised and described in Sprint 3. When specifying the trained\_checkpoint\_prefix, the following requirements must be fulfilled: The last recorded step must include a .meta, a .index, and a .data-00000-of-00001 file. (The system saves the checkpoints every fifth step. If the training is stopped in between, often the checkpoints would not include all the necessary files. In this scenario, the developer must specify an earlier stage.) Above, the last saved checkpoint was at 2615 steps. The output\_directory is the name of the frozen graph that is to be exported. In this folder, along with the checkpoints, a new .pb file gets generated called the frozen\_inference\_graph.pb. Once the folder with the necessary files is in place, the model can be used in production. This also means that the exported data becomes considerably lighter, as the additional metadata is no longer stored.

### Sprint 6 – Apply the model to classify logos

|  |  |  |  |
| --- | --- | --- | --- |
| Sprint Number | Sprint Name | Start Date | Finish Date |
| 6 | Apply model to classify logos | 20/03/2018 | 05/04/2018 |

|  |  |  |
| --- | --- | --- |
| Task Number | Details | Status |
| 1 | Test the frozen inference graph on the default Tensorflow object detection tutorial. | Complete |
| 2 | Real-time video frames are read and fed into the model as input. | Complete |

There are multiple options to test whether the training was successful. Tensorflow offers a platform -Tensorboard- to monitor the training progress as well as to help understand, debug and/or optimise the learning. It is often monitored during runtime but is also available for evaluating how well the training performed after the process is finished. There are a number of different graphs set to audit a variety of aspects such as learning rate, loss, batch, global steps, queue etc. Based on these the developer can be assured that accurate results will be achieved at the end of the training. Yet, to test the model in action, the frozen inference graph must be integrated into the an application. For this task, several solutions can be found in the API. Object\_detection\_tutorial.py was specifically designed for this purpose and served the demonstration goals perfectly. Similarly to the previous steps, certain input points had to be injected. These include the path to the checkpoints (these are automatically carried over to the folder of the graph what was specified when exported the graph (Sprint 5)), to the inference graph .pb file, the labelmap (maps indeces to the specific logo classes), defining number of classes and the path to all the images. By default, this script reads in a number of test images (must also be specified) that is analysed using the model. Initially, the plan was to detect logos on real-time video footage therefore the static image loop had to be removed. Reading webcam footage was achiveved with the usage of cv2 and some numpy functions. However, at later stages, the decision was made to re-introduce the original design because it produced better predictions. (To be evaluated in Chapter 6.) Nevertheless, both implementations are able to produce predictions and were made accessable in the project folder.

### Sprint 7 – Train the model on the Cloud

|  |  |  |  |
| --- | --- | --- | --- |
| Sprint Number | Sprint Name | Start Date | Finish Date |
| 7 | Train the model on the Cloud | 05/04/2018 | 19/04/2018 |

|  |  |  |
| --- | --- | --- |
| Task Number | Details | Status |
| 1 | Setup Google Cloud | Complete |
| 2 | Setup ML Engine | Complete |
| 3 | Create a bucket, set up the data structure | Complete |
| 4 | Start a job | Started, Incomplete |

As predicted, running a training job locally on a CPU requires much more time than what a machine with GPU does. The tested processor is an Intel(R) Core(TM) i5-5200U CPU @ 2.20Ghz with installed memory: 8.00GB. To reach ~2100 steps took four days running constantly on 98-100% memory usage. Training the model on the Cloud would expectedly shorten the amount of time to just a few hours. For this project the obvious choice cloud platform was Google Cloud. Since Tensorflow was developed by Google, the platform provides numerous tools to assist machine learning tasks such as distributed training on the ML Engine.

**Task 1-3**

The prerequisites of the training are the follows:

* The input data must be in the correct format
* A valid Object Detection pipeline is configured
* The data must be placed in a ’bucket’ on the Google Cloud Storage
* Tensorflow and the API is installed locally

The model and the structure was validated by successfully running the training on the local machine. This is recommended as cloud services can be costly. The necessary tools for this project are Machine Learning Engine and Cloud Storage. Having the data stored in the ’bucket’ required minimal configuration. An important step was to modify the configuration file restructuring the corresponding paths to the Storage bucket. Since the local training had over 2000 steps in its checkpoint, the initial goal was to introduce this file to the cloud-based training.

**Task 4**

Once a newly cloned models directory is in place, and the data is configured on the cloud the learning may begin. Regardless of the tremendous effort to debug the various errors, this part of the project remained incomplete.