# Building occupancy management solution using the TensorFlow Object Detection API

## 1 Introduction

GreenWaves has developed a people counting solution for occupancy management in smart building systems, providing real-time insights into how available space is used by employees and customers. The sensor can be used for tasks such as meeting room or cafeteria usage optimization, desk reservations and usage based cleaning.

People counting with infrared sensors offers best-in-class accuracy with total compliance to privacy-related regulations for indoor environments. GAP processors provide a combination of computing ability for AI and low energy operation that enables this type of application.

As part of our development process, we needed to train an optimized neural network with a single shot detector SSD backend. The TensorFlow Object Detection API comes with a number of prepackaged backbone models, but we wanted to design something more optimized for our detection task. We aimed to

- reduce memory,
- reduce complexity, and
- reduce power consumption

In this document, we will show how we carried this out and how a custom network design can still leverage all the backend SSD creation offered by the TensorFlow Object Detection API. We hope this will allow you

- **become familiar with an object detection API like the one provided by TensorFlow.**
- **learn how to modify the API with respect to your custom specifications (i.e., model structure).**
- **learn how to employ the API for custom solutions such as occupancy management.**
- **learn how to generate optimized code for running your solution on GreenWaves' GAP processors.**

## 2 Object detection API

Constructing, training, and deploying machine learning models for the localization and identification of multiple objects is a challenging task. To make this easier, we attempted to leverage the TensorFlow Object Detection API, an open source framework for object detection built on top of TensorFlow. The API involves a group of useful object detection methodologies including

- Single Shot MultiBox Detector (SSD)
- CenterNet
- RCNN
- EfficientDet
- ExtremeNet

Please use these links (TensorFlow 2 Detection Model Zoo, TensorFlow 1 Detection Model Zoo) to view a full list of object detection methodologies supported by the API. It is clear that these solutions have different network architecture, training, and optimization strategies. If you are interested in finding out more, you can read this article for more details on different frameworks along with their advantages and disadvantages. Although these frameworks exhibit different characteristics, all employ deep convolutional neural networks (CNNs) to extract high-level features from the input images, called ***backbone*** models. In fact, it has become normal practice to employ and adapt the modern state-of-the art CNNs for feature extractor backbones. This can be achieved by removing the final fully connected classification layers from a CNN, leaving a deep neural network that can be used to extract semantic meaning from the input image without changing its spatial structure.

The following are some useful CNN structures that can be used for backbone models of detectors:

- VGGNet
- MobileNet
- ResNet
- GoogleNet
- DenseNet
- Inception

TensorFlow already provides a collection of detection models backboned to pre-trained CNNs on datasets like COCO and ImageNet. These models can be used for out-of-the-box inference if the target categories are already included in these datasets. Otherwise, they can be used to initialize the model when training on new datasets.

In the rest of this article, we will focus on the SSD object detection algorithm and show you how to use the TensorFlow Object Detection API to develop your own detection network.

## 2.1 Single Shot MultiBox Detector (SSD)

An SSD network has two principal components: a backbone model and an SSD head. As explained earlier, the backbone is typically a CNN model that may be inherited from a state-of-the-art deep model trained on datasets like Imagenet and COCO. The SSD head consists of one or more convolutional layers added to the end of the backbone network where object bounding box classification takes place. The SSD head layers predict the offsets and associated confidence scores to a designed set of default bounding boxes of different scales and aspect ratios (Figure 1).

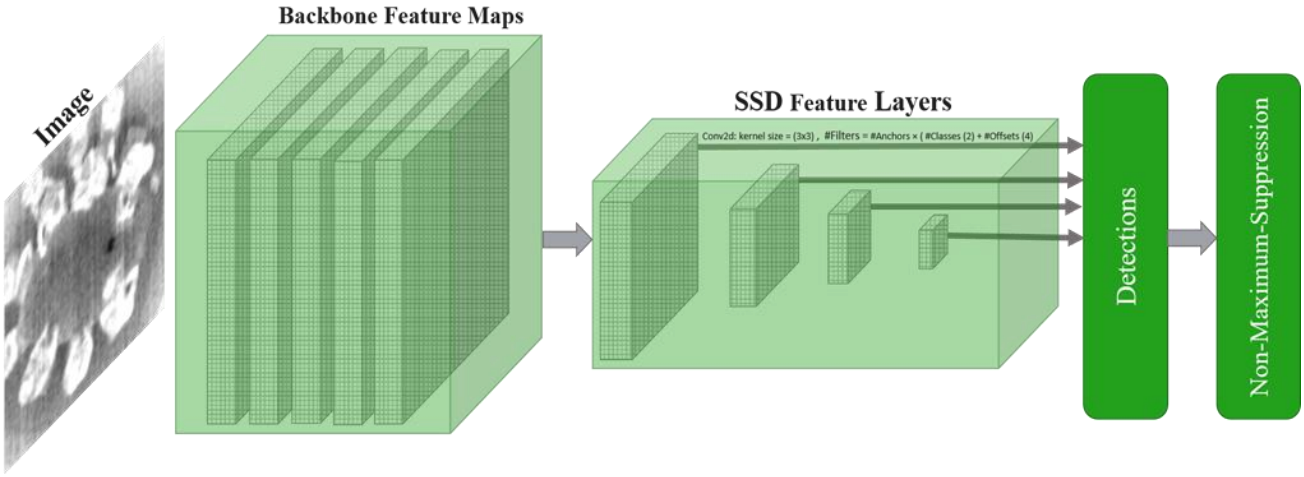


*Figure 1. An SSD model structure that adds several feature layers to the end of a base network that predicts the offsets to default boxes of different scales and aspect ratios and their associated confidences*

As can be seen in Table 1, TensorFlow provides various SSD heads backboned to a collection of pre-trained models to facilitate the training process.

| MODEL NAME | SPEED (MS) | COCO MAP |
|---|---|---|
| SSD MOBILENET V1 FPN 640X640 | 48 | 29.1 |
| SSD MOBILENET V2 FPNLITE 320X320 | 22 | 22.2 |
| SSD MOBILENET V2 FPNLITE 640X640 | 39 | 28.2 |
| SSD RESNET50 V1 FPN 640X640 (RETINANET50) | 46 | 34.3 |
| SSD RESNET50 V1 FPN 1024X1024 (RETINANET50) | 87 | 38.3 |
| SSD RESNET101 V1 FPN 640X640 (RETINANET101) | 57 | 35.6 |
| SSD RESNET101 V1 FPN 1024X1024 (RETINANET101) | 104 | 39.5 |
| SSD RESNET152 V1 FPN 640X640 (RETINANET152) | 80 | 35.4 |

In the table, the ***speed*** refers to the running time in *ms* per input image, which includes all corresponding preprocessing and post-processing steps. It should be mentioned that the runtime values highly depend on hardware configuration, and these values are produced using a unique computer but are useful as a relative measure of latency.

# 3 SSD solution deployment using the API

Using pretrained models as an SSD backbone eases the training process but puts constraints on the network structure. To enable efficient inference on the edge, we need to train a custom CNN solution for applications such as infrared human detection that only require a small backbone model with a relatively restricted number of parameters.

In the next section, we will show how you can modify the TensorFlow Object Detection API in order to construct any

custom SSD model. If you are already familiar with the theoretical concept of a Single Shot MultiBox Detector, then this section will provide you with a concrete example that will allow you to develop any custom SSD model using the API.

## 3.1 API model structure

All models under the TensorFlow Object Detection API must implement the ***DetectionModel*** interface; for more details, you can take a look at the file defining the generic base class for detection models in the API:

- API: object_detection/core/model.py

At a high level, detection models receive input images and predict output tensors. At training time, output tensors are directly passed to a specified loss function while at evaluation time, they are passed to the post processing function, which converts the raw outputs into actual bounding boxes. The Object Detection API follows this structure. The model you want to train should include the five functions below:

- **Preprocess** applies any preprocessing operation to the input image tensor. This could include transformations for data augmentation or input normalization.
- **Predict** produces the model's raw predictions that are passed to the corresponding loss or post processing functions (e.g., Non-Maximum Suppression).
- **Postprocess** converts raw prediction tensors into appropriate detection results (e.g., bounding box index and offset, class scores, etc.).
- **Loss** defines a loss function that calculates scalar loss tensors over the provided ground truth.
- **Restore** loads checkpoints into the TensorFlow graph.

Depending on whether you are training or evaluating the network, a batch of input images passes through a different sequence of steps, as depicted in Figure 2.

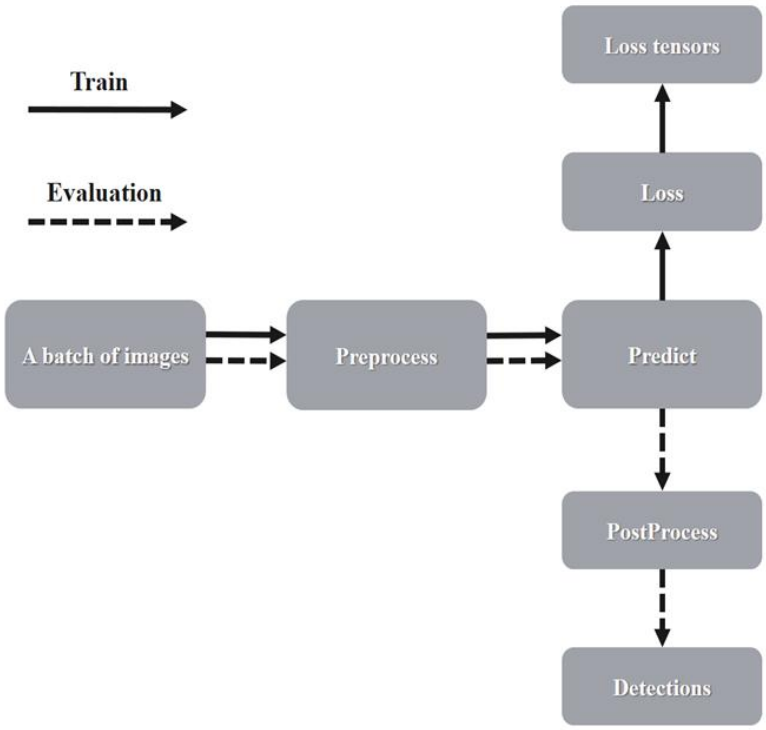


*Figure 2.*

## 3.2 API object detection models

To allow the construction of ***DetectionModels*** for various object detection methodologies (i.e., SSD, CenterNet, RCNN, etc.), different ***meta-architectures*** are implemented by the TensorFlow Object Detection API. The idea behind ***meta-architectures*** is to provide a standard way to create valid ***DetectionModels*** for each of the object detection methodologies**.** All object detection ***meta-architectures*** can be found at the following link:

- API: object_detection/meta_architectures

In the case of custom models, you have the option of implementing a complete ***DetectionModel*** following a specific ***meta-architecture***. However, instead of defining a model from scratch, it is possible to create only a feature extractor that can be employed by one of the pre-defined ***meta-architectures*** to construct a ***DetectionModel***. It should be emphasized that meta-architectures are classes that define entire families of models using the DetectionModel abstraction.

## 3.3 The SSD meta-architecture API

Before describing the stages in the development of a custom SSD model, it is important to establish an understanding of the details of the SSD meta-architecture. As you can see in the example SSD model, there are three principal parts to an SSD model:

- SSD feature maps
- Prediction layers (i.e., classes and offsets)
- Post processing layers

When constructing your model, the Object Detection API uses a model configuration file to automatically create prediction and post processing layers. The configuration contains the anchor generator (e.g., aspect ratios and scales of the default bounding boxes), box predictors (e.g., convolution layer hyper parameters), and post processing (e.g., iou and score thresholds) parameter values. However, SSD feature maps are created by employing pre-constructed feature extractor models. The full list of SSD feature extractor models can be found at

- API: object_detection/models

We can choose an appropriate feature extractor model from the pre-constructed models in the configuration file (i.e., feature_extractor). However, this requires us to know the mappings from model names to their pre-defined structures before changing the configuration file. This mapping can be found at

- API: object_detection/builders/model_builder.py

Also, a number of sample configuration files are provided in the following API:

- [API: object_detection/configs/tf2]

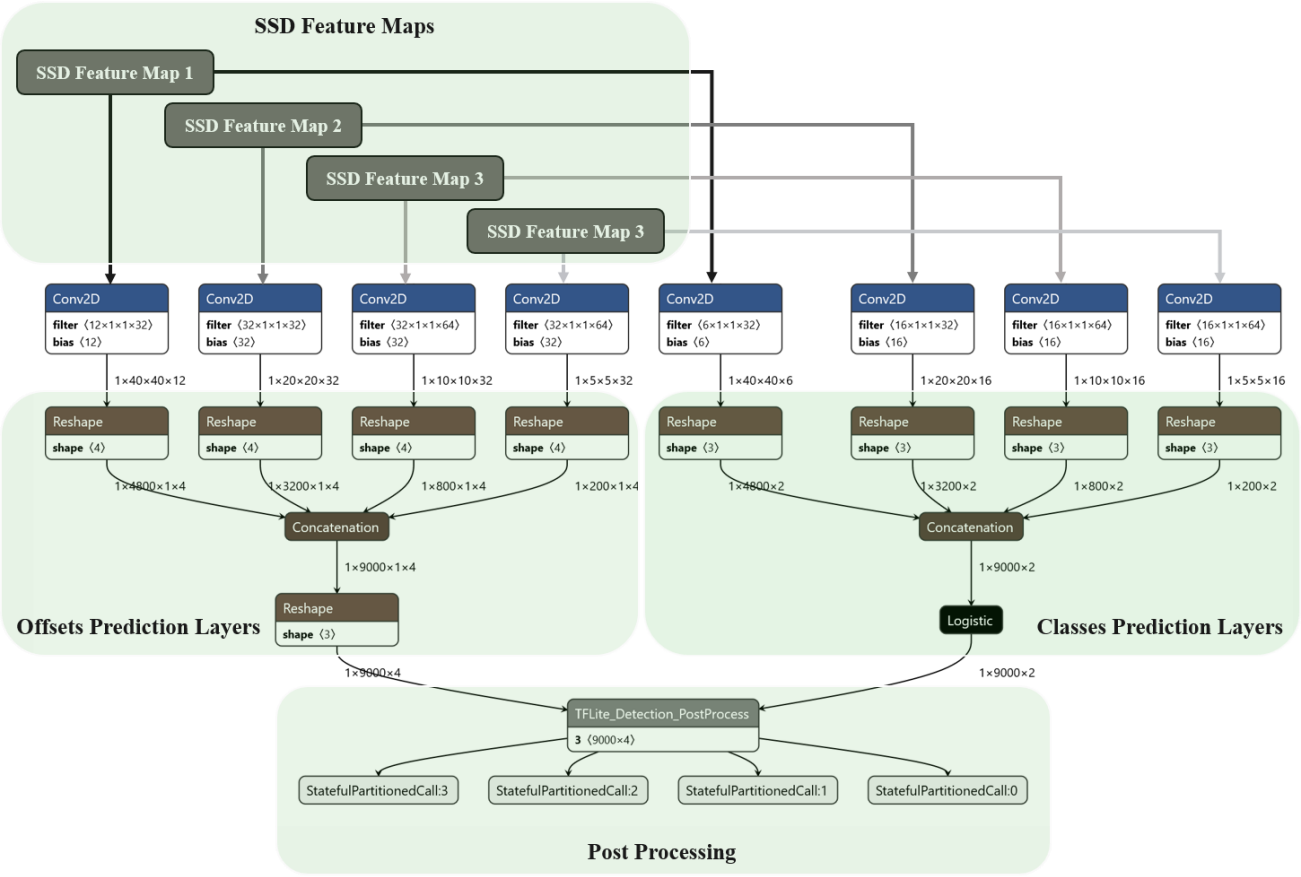


*Figure 3. A typical SSD model constructed by the API*

An SSD meta-architecture (SSDMetaArch) requires a feature extractor (SSDFeatureExtractor) to automatically construct appropriate class and offset prediction layers according to the configuration parameters (i.e., number of classes, scales, and aspect ratios). The general framework is summarized in Figure 4.


*Figure 4. SSD DetectionModel construction framework*

## 3.4 Custom SSD model design

Now, let's look at how we can construct a custom SSD model using API meta-architectures. While we can implement a custom SSD detection model from scratch, the previous sections have shown that the construction of a custom model can be achieved via the definition of an appropriate feature extractor model. The API will automatically create prediction and post processing layers using the configuration parameters. We only need to construct an SSD feature extractor through the SSDFeatureExtractor class. The custom feature extractor can be added to the mapping of the feature extractor in

- API: object_detection/builders/model_builder.py

A visual representation of all the elements of the SSDFeatureExtractor class necessary to define a valid SSD feature extractor is shown in Figure 5.

- API: object_detection/meta_architectures/ssd_meta_arch.py

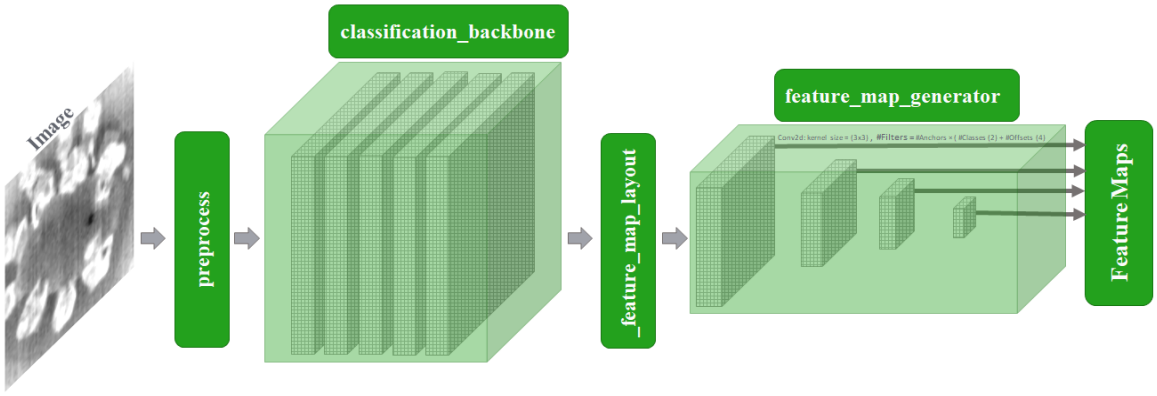


*Figure 5. A valid SSD feature extractor structure*

In the next section, we explain the principal components of the SSDFeatureExtractor and show how one can construct or adapt a feature extractor for any kind of application.

### 3.4.1 Preprocess

This defines the preprocessing operation that normalizes input images for the classification backbone.

```
def preprocess(self, resized_inputs):
    """SSD preprocessing.

    Maps pixel values to the range [-1, 1].

    Args:
    resized_inputs: a [batch, height, width, channels] float tensor
    representing a batch of images.

    Returns:
    preprocessed_inputs: a [batch, height, width, channels] float tensor
    representing a batch of images.
    """
    return (2.0 / 255.0) * resized_inputs - 1.0
```

### 3.4.2 Classification_backbone

The classification backbone is the network structure for the extraction of basic feature maps from the preprocessed inputs.

```
# input layer

Input = tf.keras.layers.Input(shape=(input_shape[1], input_shape[2], input_shape[3]), batch_size = input_shape[0])


#construct a base model without any prediction/classification layers

Output1 =… #a layer output with a specific layer name like 'mp1'
Output2 =… #a layer output with a specific layer name like 'mp2'
Output3 =… #a layer output with a specific layer name like 'mp3'


# base feature maps being used by feature map generator
Base_feature_maps = [Output1, Output2, Output3]


self.classification_backbone = tf.keras.Model(inputs = Input, outputs = Base_feature_maps)
```

### 3.4.3 _feature_map_layout

This is a dictionary that determines which basic feature maps are being used to generate SSD feature maps by feature_map_generator.

```
# Self._num_layers, self._use_dpthwise, and self._use_explicit_padding are
# extracted from the config file.

 self._feature_map_layout = {
        'from_layer': ['mp1', 'mp2', 'mp3', ''][:self._num_layers],
        'layer_depth': [-1, -1, -1, 256][:self._num_layers],
        'use_depthwise': self._use_depthwise,
        'use_explicit_padding': self._use_explicit_padding,
        }
```

In this example, the SSD model would have four prediction layers; the first three layers are provided by mp1, mp2, and mp3, and the API will automatically create the fourth (' ') with 256 features. Remember that the API will use your last feature map ('mp3') as input for the fourth one.

It is very important to keep in mind that there should be a correspondence between the layers' names provided in the _feature_map_layout and the backbone_classification model, as the feature map generator takes those layers' outputs as its inputs for the construction of feature maps.

### 3.4.4 feature_map_generator

This is used to construct SSD feature maps from the features determined in the feature_map_layout.

```
# Self._depth_multiplier, self._depth_multiplier, self._min_depth,
# self._conv_hyperparams and self._freeze_batchnorm are extracted from the
# config file.


    self.feature_map_generator = (
        feature_map_generators.KerasMultiResolutionFeatureMaps(
         feature_map_layout=self._feature_map_layout,
         depth_multiplier=self._depth_multiplier,
         min_depth=self._min_depth,
         insert_1x1_conv=True,
         is_training=self._is_training,
         conv_hyperparams=self._conv_hyperparams,
         freeze_batchnorm=self._freeze_batchnorm,
         name='FeatureMaps'))
```

The feature map generator has a function in its API to create feature maps automatically so there is no need for further coding.

### 3.4.5 Feature maps

SSD feature maps are created for each input image as shown below:

```
'Step1) pass preprocessed input to the classification backbone'
        image_features = self.classification_backbone(
        ops.pad_to_multiple(preprocessed_inputs, self._pad_to_multiple))


        'Step2) construct feature maps from image features.'
        feature_maps = self.feature_map_generator({
        'mp_1': image_features[0],
        'mp_2': image_features[1],
        'mp_3': image_features[2],
        'mp_4': image_features[3]})
```

## 3.5 Custom model embedding

Following the implementation of the instructions in Section 4 and the construction of a custom SSD feature extractor (e.g., ssd_custome_keras_feature_extractor.py), first, we have to ensure that the file has been added to the API models folder at

- API: object_detection/models

Second, this custom feature extractor can be added to the mapping in which all feature extractor definitions are provided, and we can access them within the pipeline configuration file using their corresponding keys. To do this, open the API model builder at API: object_detection/builders/model_builder.py and add following lines:

```
 from object_detection.models.ssd_custome_keras_feature_extractor import
SSDMCustomKerasFeatureExtractor

"""for models created using keras API"""
SSD_KERAS_FEATURE_EXTRACTOR_CLASS_MAP = {
        ...

        'ssd_custom': SSDMCustomKerasFeatureExtractor,

        ...
        }
```

## 3.6 API installation

After applying all modifications related to the custom model design, follow the normal steps for the installation of the API provided on the TensorFlow website. After the installation of the API, the custom SSD feature extractor can be accessed within the configuration file:

```
model {
  ssd {
        inplace_batchnorm_update: boolean
        freeze_batchnorm: boolean
        num_classes: int
        box_coder {...bounding boxes coder parameters...}
        matcher {...argmax matcher parameters...}
        similarity_calculator {...similarity measure and its corresponding parames...}
        encode_background_as_zeros: true

        anchor_generator {...SSD anchor generator parameters like aspect ratio, etc...}
        image_resizer {... image resizer...}
        box_predictor {... predictor heads parameters ...}
   feature_extractor { type: 'ssd_custom', and other info}
        loss { ...model loss functions...}
        post_processing {...postprocessing parameters}
}
```

# 4 Occupancy management data preparation

Some infrared images used for training the human detection model are shown in Figure 6. Images are acquired by various sensors installed at different locations and heights. It should be noted that some acquisitions are blurry due to bad focus adjustment after sensor installation. It would be better to adjust focus according to the sensor height, but our model will learn to work around this.

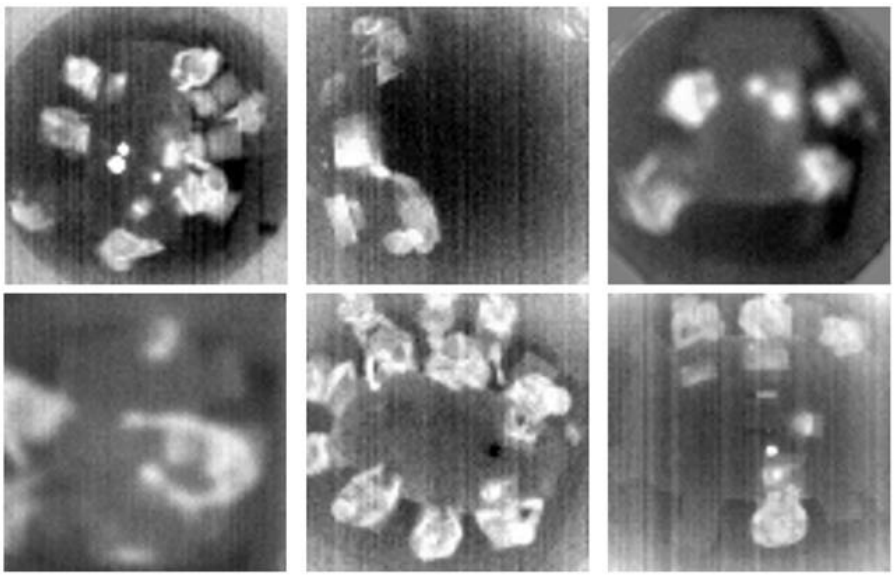

*Figure 6. Infrared images of humans at various locations*

## 4.1 Annotations

Human annotations are provided in the Pandas DataFrame structure exported to csv files, which include image filenames and their corresponding bounding boxes. An example of this information is provided in the table below.

| image_name | xmin | xmax | ymin | ymax | class_id |
|---|---|---|---|---|---|
| acqui3_imag_10179.png | 16 | 37 | 54 | 77 | 1 |
| acqui3_imag_3723.png | 12 | 33 | 4 | 31 | 1 |
| acqui3_imag_3723.png | 59 | 79 | 48 | 71 | 1 |
| acqui2_image_1509.png | 48 | 66 | 64 | 78 | 1 |
| acqui2_image_1509.png | 61 | 73 | 60 | 73 | 1 |
| acqui2_image_1509.png | 50 | 62 | 31 | 45 | 1 |
| acqui2_image_1509.png | 14 | 32 | 38 | 50 | 1 |
| acqui2_image_1509.png | 54 | 70 | 15 | 30 | 1 |
| acqui2_image_1509.png | 17 | 31 | 14 | 29 | 1 |
| acqui2_image_1509.png | 36 | 47 | 3 | 13 | 1 |

In Figure 7, for each input image, corresponding ground truth bounding boxes are drawn where humans are present in the frames.

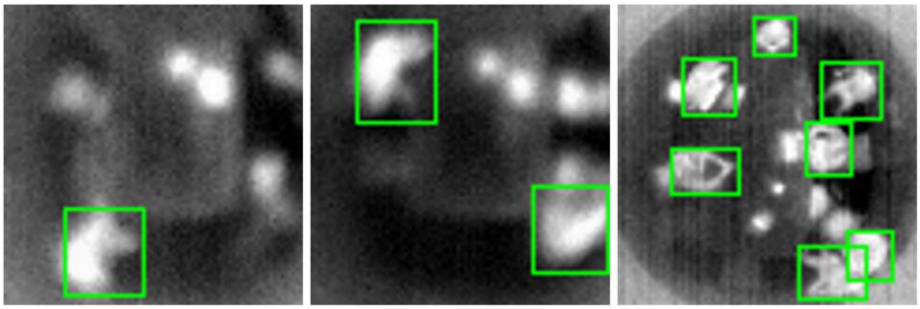

*Figure 7. Ground truth bounding boxes are drawn around corresponding images*

## 4.2 API data preparation

The id of the classes start from 1, and the class id of 0 is reserved for the background context. In the human detection model, the labels map is as follows:

```
item {
  id: 1
  name: 'human'
}
```

*In summary, the corresponding* `tf_example` *for each image in the dataset* is created and stored in the dataset's TFRecord file (Figure 8).


Figure 8. TFRecord file generation

# 5. Summary

In this document, we provided an overview on how to train and optimize a neural network for occupancy management applications, leveraging all the backend SSD offered by the TensorFlow Object Detection API.

To enable easy porting of custom NNs on GAP, we have developed GAPflow, a set of tools released by GreenWaves Technologies that allows users to accelerate the deployment of NNs on GAP while ensuring high performance of and low-energy consumption on GAP processors. The GAPflow toolset assists programmers in achieving short time-to-prototype of DL-based applications by generating GAP-optimized code based on the provided DL model, and it fully supports the importation of detection models created using the Tensorflow Object Detection API. Watch our tutorial here.